5-2010

Improved Framework for Fast and Efficient Memory-based Frame Data Reconfiguration for Multi-row Spanning Designs on Field Programmable Gate Arrays

Rohan Sreeram
Utah State University

IMPROVED FRAMEWORK FOR FAST AND EFFICIENT MEMORY-BASED FRAME DATA RECONFIGURATION FOR MULTI-ROW SPANNING DESIGNS ON FIELD PROGRAMMABLE GATE ARRAYS

by

Rohan Sreeram

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Aravind Dasu Major Professor	Prof. Paul Israelsen Committee Member
Dr. Fon Brown Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2010

Abstract

Improved Framework for Fast and Efficient Memory-based Frame Data Reconfiguration for Multi-row Spanning Designs on Field Programmable Gate Arrays

by

Rohan Sreeram, Master of Science

Utah State University, 2010

Major Professor: Dr. Aravind Dasu
Department: Electrical and Computer Engineering

Reconfigurable computing is an evolving paradigm in computer architecture where the ability to load different designs onto a field programmable gate array (FPGA) at execution time has proven useful in adapting FPGA prototypes to a wide range of applications. Reconfiguration techniques can be primarily categorized as Partial Dynamic Reconfiguration (PDR) and Partial Bitstream Relocation (PBR). PDR involves reconfiguring a single Partial Reconfiguration Region (PRR) with a partial bitstream, while PBR is targeted at reconfiguring multiple PRRs on the FPGA with a partial bitstream. Previous techniques have primarily focused on using either slower off-chip memory or on-chip memory-based solutions to store the partial bitstream, and then reconfigure a PRR on the FPGA. Another technique called Accelerated Relocation Circuit (ARC) provides a more efficient method where a PRR (active bitstream) is used to relocate to other PRRs on the fly using minimal on-chip memory. This thesis proposes a novel technique for Memory-based Frame Data Reconfiguration (M-FDR) of multi-row PRRs. ARC hardware was re-architected to provide an improved frame data reconfiguration framework, called Accelerated Memory-based Reconfiguration Circuit (AMRC) for use in MBR scenarios. A performance prediction model is also proposed that confirms the speedup achieved by AMRC, in comparison to ARC and

earlier methods. This technique was found to be 26.6% faster than ARC in PRR-PRR relocation. In comparison to other relocation techniques like Bit Relocation Filter (BiRF), AMRC provides a speedup of 231x. The AMRC method was also able to dynamically parallelize multi-row designs with an average context switching time of 0.37 ms.

(75 pages)

To my family and loved ones.

Acknowledgments

I would like to thank Dr. Aravind Dasu for his guidance and suggestions that greatly helped me in accomplishing this thesis. His expertise in the field of reconfigurable computing is remarkable and working under him was a highly rewarding experience. I would also like to thank my committee members, Prof. Paul Israelsen and Dr. Fon Brown, for extending their support. I thank Ram Chandra Kallam for helping me understand his previous work in frame data relocation. I would like to thank Hari Samala and Vaibhav Ghadiok for their invaluable comments during the documentation process. I also thank my friends Tarun, Ajit, Shant, Aditya, Swagat, Chandana, Swati, Meghana, Rachit, Shivani, Deepti, Akbar, Balaji, Gurpreet, and all my other friends for their support and wishes that kept me going all the time. I thank my parents, sister, and brother-in-law for believing in me and for all their sacrifices, support, and patience during the entire course of my study.

Rohan Sreeram

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter the motivation for a fast and efficient memory-based reconfiguration technique in Xilinx Field Programmable Gate Arrays (FPGAs) is discussed. This is followed by the principal contributions of this thesis, and finally an overview of the content presented in this thesis.

1.1 Motivation

A major design constraint during Partial Bitstream Relocation (PBR) is the need to generate a different partial bitstream for each Partial Reconfiguration Region (PRR), even if the same circuit is being copied to all the target regions. The storage of multiple versions of the partial bitstream is not a viable option, since the storage needs of a circuit increase linearly with the number of PRRs. Many techniques use the off-chip flash memory or on-chip Block RAMs (BRAMs) to store the partial bitstream. The partial bitstream is then typically transformed for the target PRR, using either a hardware-based implementation or relocation software running on an on-chip processor. Techniques proposed by Becker et al. [1], Kalte et al. [2], H. Kalte and M. Porrmann [3], Corbetta et al. [4], Carver et al. [5] suffer from performance bottlenecks either attributed to the use of software-based relocation, or hardware-based relocation filters that rely on off-chip memory for partial bitstream storage.

Sudarsanam et al. [6] proposed a revolutionary technique in PRR relocation, using the Accelerated Relocation Circuit (ARC), where the need for bitstream storage is totally eliminated. The relocation in ARC is done by relocating the actual frame data inside the partial bitstream. This technique is called Frame Data Relocation (FDR). FDR has significant performance advantages as compared to traditional PBR, because in addition

to the frame data being relocated, the overhead to continually perform Cyclic Redundancy Check (CRC) calculations is also eliminated.

Although the use of the FDR technique in ARC has largely reduced the relocation time without storing the entire partial bitstream, it does not support reconfiguration scenarios where designs need to frequently switch between memory and the PRRs. Regeneration of designs at run-time increases resource utilization by reducing idle circuit time, in addition to accomodating a larger subset of reconfigurable designs on the FPGA chip. This has been found to be useful in applications like dynamically parallelizable systolic array architectures [7] that can switch to different Processing Element (PE) designs at run-time, based on change in computational requirements. Another useful application for regeneration of designs has been explored by Sreeramareddy et al. [8] using self healing reconfigurable architectures for mission-critical space systems.

Real-world designs owing to their resource usage often tend to occupy multiple Horizontal Clock (HCLK) rows and columns in the FPGA. ARC does not support such large designs where a design could span multiple HCLK rows.

The use of on-chip memory has become critical to enable faster reconfiguration of a PRR from memory. Memory-based reconfiguration involves storing bitstream information in on-chip memory, so that it can be configured into a PRR on demand. This is particularly helpful in partial reconfiguration scenarios where only a subset of the designs is active at any given point in time, and will be replaced by a different subset of designs based on user demand. The on-chip memory utilization is an important design constraint to be considered when designing memory reconfiguration circuits. This is because the on-chip memory used by the memory-based reconfiguration circuit is limited by the amount of BRAM resources available for use by the designs.

1.2 Principal Contributions

The overall goal of this thesis work is to design and develop a frame data reconfiguration framework which can achieve the following:

1. Accommodate PRRs that span across several HCLK rows and columns;
2. Provide a mechanism to rapidly regenerate designs from on-chip memory without incurring the latency present in off-chip memories;
3. Efficient use of on-chip memory to store reconfigurable designs.

1.3 Thesis Overview

The following chapter (Chapter 2), discusses the basics of FPGAs, the Xilinx design flow methodology, and explains the Early Access Partial Reconfiguration (EAPR) which is the partial reconfiguration methodology specified by Xilinx. Following this a formal introduction to partial dynamic reconfiguration and partial dynamic bitstream relocation has been provided. Finally the chapter ends with a literature review of previous work on PBR techniques. Chapter 3 is a detailed explanation of the multi-row PRR-PRR relocation algorithm. Chapter 4 discusses the proposed Memory-based Frame Data Reconfiguration (M-FDR) technique and the performance prediction model. In Chapter 5, the hardware design internals of the Accelerated Memory-based Reconfiguration Circuit are described, along with a performance comparison with ARC. This is followed by a discussion of the test case scenarios used for Accelerated Memory-based Reconfiguration Circuit (AMRC), and the results obtained in Chapter 6. Finally, Chapter 7 gives the conclusion and future work for this thesis.

Chapter 2

Background and Related Work

2.1 Field Programmable Gate Array (FPGA)

An FPGA is a collection of several configurable logic memory cells (typically Static Random Access Memory (SRAM)), also called a gate array, which can be reconfigured to implement different designs. The reconfigurability in the design of an FPGA is defined by the user, typically expressed in a Hardware Description Language (HDL). The hardware description of a design is then synthesized into a gate-level description (also called a netlist). All the input netlists are merged into a single logical design in terms of basic device primitives. This is followed by a mapping of the logic and device primitives to the corresponding Configuration Logic Blocks (CLBs) and Input/Output blocks (IOBs) on an FPGA. Finally these hardware blocks are placed and routed and the corresponding bit stream for the circuit is generated. We will look at the FPGA design flow in greater detail in the later sections of this chapter.

Major FPGA vendors include Altera and Xilinx. This section presents the discussion of FPGA architectures and is restricted to the Xilinx Virtex-4 family of FPGAs as this was the target platform chosen for implementation in this thesis. FPGA architectures are built around the following building blocks: Configuration Logic Blocks (CLBs), Block Random Access Memory (BRAM), Input/Output Blocks (IOBs), Global CLocKing logic (GCLK), and a configurable switch matrix for routing. Figure 2.1 depicts these building blocks and their interconnectivity in a typical Virtex-4 FPGA.

These building blocks are typically arranged in a grid fashion with each block interconnected to the other blocks via switch matrices, which provide the configurable routing logic between them.

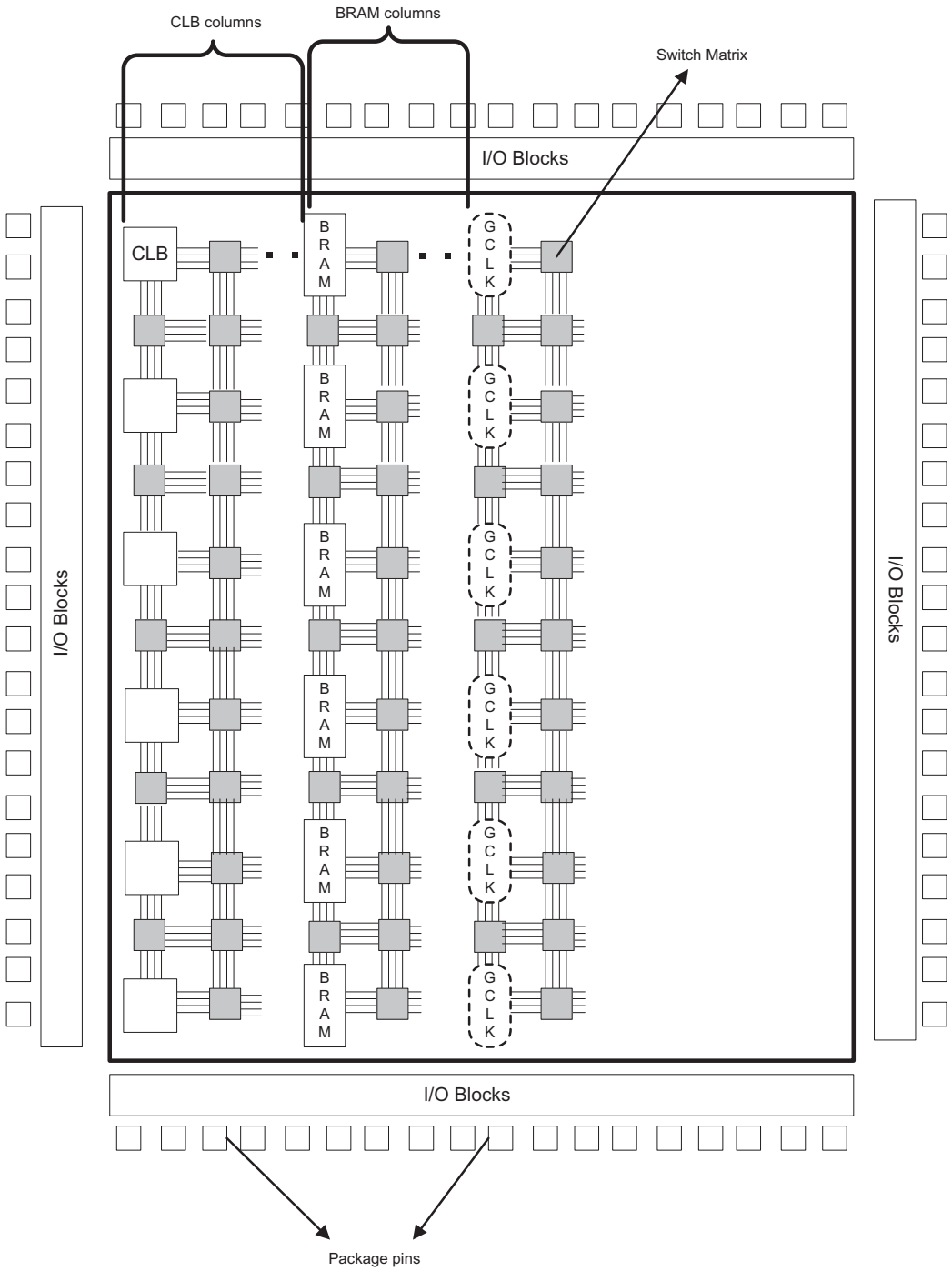

Fig. 2.1: A typical Virtex-4 FPGA architecture.

CLBs form the primary logic resource in an FPGA. They are used to store both combinatorial and sequential logic circuits. Each CLB in Virtex-4 comprises of four slices (as shown in Fig. 2.2).

Each slice in turn consists of two Look Up Tables (LUTs) and two storage elements. Each LUT is a four-input function generator that can essentially realize any circuit with four inputs by storing its behavior in the form of a truth table. Since a LUT typically has four inputs, the configuration data that each LUT holds is 16 bits which are the 16 outcomes for the different input combinations fed into a 4-input combinatorial circuit being implemented. Each storage element can be configured to function as an edge-sensitive flip-flop or a level-sensitive latch. Figure 2.3 [9] provides a simplified view of a slice's internals that include a pair of four-input LUTs and a pair of storage elements. The multiplexer MUXF5 in the figure is used to combine the outputs of the two LUTs in order to form 5-input combinatorial circuits. The multiplexer MUXFX is used to select outputs from neighboring slices. Other multiplexers shown are not user-controlled and the FPGA programming tools configure them when generating a bitstream.

BRAMs are dedicated configurable blocks of on-chip memory which hold 18 Kbits of data on a virtex-4 FPGA. Xilinx provides tools like Block Memory Generator (BMG) using which BRAMs can be customized as single/dual-port RAM/ROM modules. BRAMs are organized as an array of fast SRAM cells with additional circuitry to fetch data synchronously similar to reading a register file. The edge-triggered logic inside BRAMs makes them a faster alternative to asynchronously fetching data from LUT-based RAMs, where timing issues are involved. BRAMs provide two symmetrical and totally independent input ports that can be used to access the data inside a BRAM. BRAMs also support different aspect ratios such as 16Kx1, 8Kx2, up to 512x36, by varying the address and data widths. BRAMs are therefore a flexible, fast and efficient on-chip memory solution especially useful for applications that have large memory requirements.

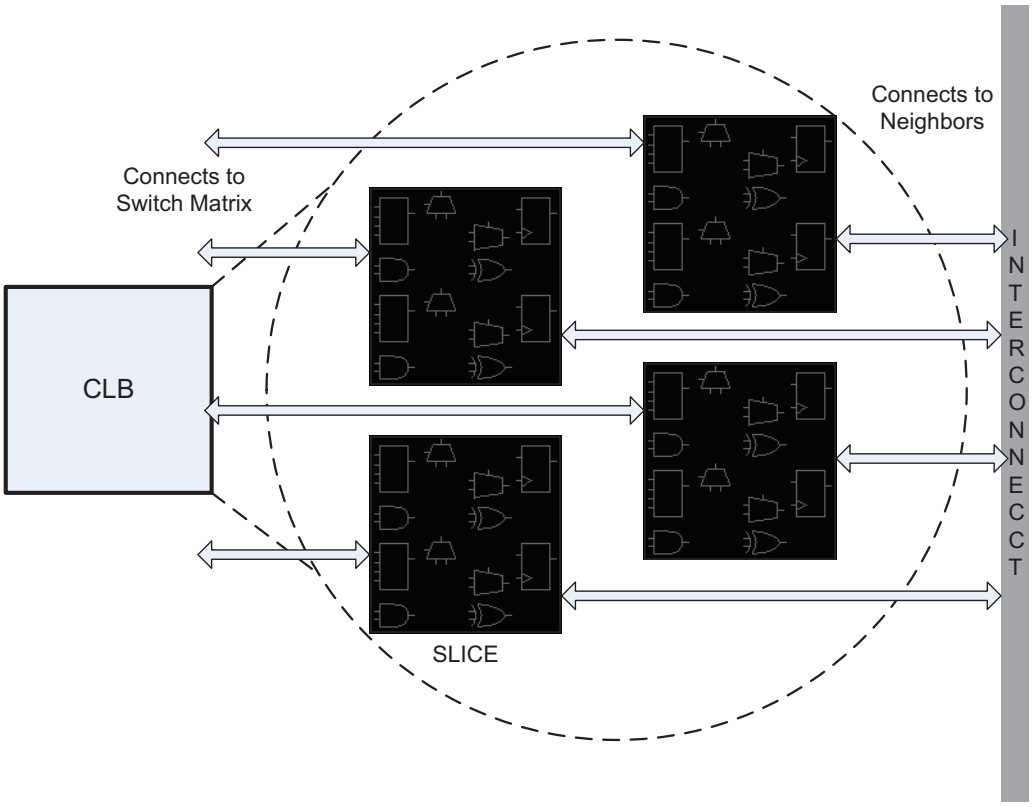

Fig. 2.2: CLB internal organization.

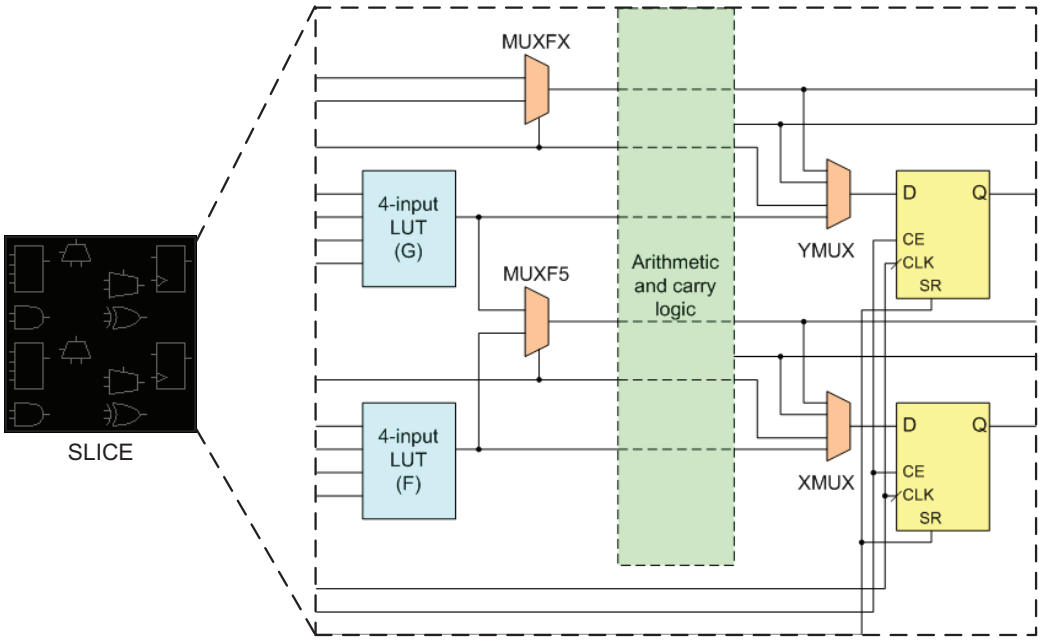

Fig. 2.3: Simplified view of a slice's internals.

2.2 Xilinx FPGA Design Flow

The Xilinx FPGA design flow as mentioned by Xilinx [10] involves design entry, design synthesis, design implementation, bitstream generation, and finally programming the FPGA with the bitstream. Figure 2.4 depicts these different stages and how they are interdependent on each other.

Design entry involves the user specifying the design, either using a Xilinx schematic editor or a Hardware Description Language (HDL). Once the design has been created, the Xilinx tools synthesize it into a netlist. The netlist is a collection of logic gates and interconnect that represent the user design logic, but does not contain the circuit's timing details. It is important that the correctness of the design be verified at this stage by performing a functional simulation. The functional simulation helps verify that the user logic in the design is consistent with the design requirements. The timing results of the design are available only after the design implementation phase.

After the design implementation has completed, the timing simulation helps verify that the design is able to run correctly, after accounting for all wire and component delays. The Xilinx tools can now be used to generate the final bitstream that the FPGA can be programmed with. This bitstream contains the configuration data in the form of a series of command and data word pairs that instruct the FPGA's configuration hardware to program it.

2.3 Virtex-4 FPGA Configuration Memory Layout

Virtex-4 configuration memory is organized as frames, which are the smallest addressable data units in configuration memory. Therefore the smallest amount of data read or written to an FPGA is a frame. Each frame in a Virtex-4 family is 1312 bits, or 41 32-bit words in length. Every frame in the FPGA can be accessed via a unique address, termed as its frame address. The frame address format is given in Fig. 2.5.

In order to configure the FPGA, one needs to program the Frame Address Register (FAR) with the correct frame address followed by the frame data to be read or written. The frame address bits are organized as follows.

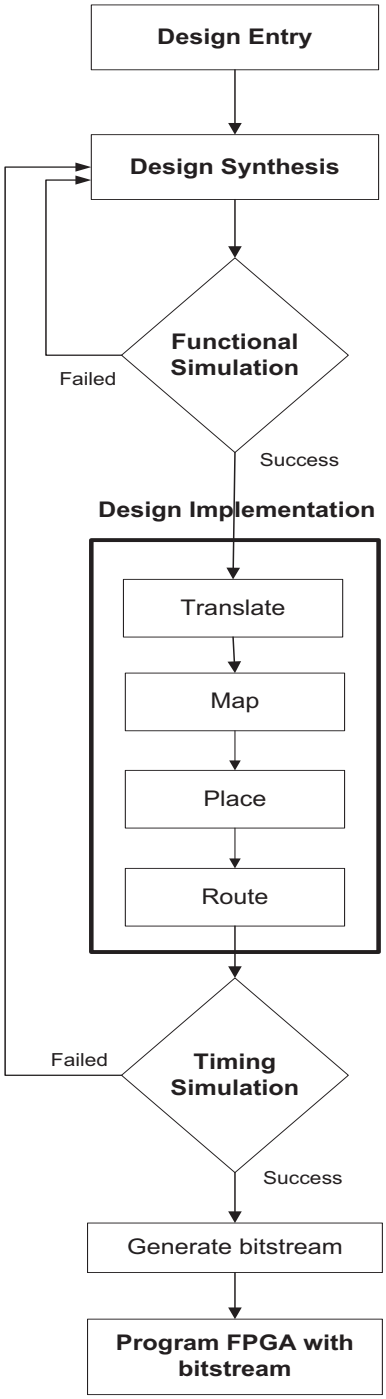

Fig. 2.4: Xilinx FPGA design flow.

Address Type	Bit Index	Description
Top/Bottom Bit	22	Select between top-half rows and bottom-half rows.
Block Type	21:19	Block types are, CLB/IO/CLK (000), block RAM Interconnect (001), block RAM content (010), CFG_CLB (011), and CFG_BRAM (100). A normal bitstream stops at type 010.
Row Address	18:14	Selects a row of frames, for example, a row of 16 CLBs in height, with an HCLK row in the middle. The row addresses increase away from the middle (in both top and bottom).
Column Address	13:6	Selects a major column, such as a column of CLBs. Column addresses start at 0 on the left and increase to the right.
Minor Address	5:0	Selects a memory-cell address line within a major column.

Fig. 2.5: Frame addressing format.

2.3.1 Top/Bottom

This bit defines whether the frame is located in the top or bottom half of the FPGA chip. An important characteristic of which half of the chip a frame is located in, is that the frames in the top half are always mirror images of the frames in the bottom half.

2.3.2 Block Type

The block type is a 3-bit number that defines the type of FPGA component the frame belongs to. CLB, IOBs, and Digital Signal Processing (DSP) blocks belong to a single block type, while BRAM interconnect and BRAM content are of separate block types. The different block types and their mappings are given in Fig. 2.5.

2.3.3 Row Address

The Virtex-4 FPGA is divided into six horizontal rows, three in the top half and three in the bottom half of the FPGA. These rows are called Horizontal Clock (HCLK) rows, since each row represents a separate clock region. Each set of rows in the top and bottom half of the FPGA is numbered incrementally from 0 to 2 starting from the middle of the FPGA. This is better illustrated in Fig. 2.6.

2.3.4 Column Address

This field specifies the column number for the specific block type of the given frame. Different block types have different column addressing schemes increasing from 0 from the left to right of the FPGA chip. Figure 2.6 shows the column addressing for block type 000. This is also known as the Major Column Address.

2.3.5 Minor Address

Each column in the FPGA consists of several frames. The number of frames within each column depends on the type of FPGA resource. Table 2.1 summarizes the number of frames per column (or the number of minor columns), for each resource type.

2.4 Partial Dynamic Reconfiguration (PDR)

FPGA reconfiguration is the process of reconfiguring an FPGA either completely (full reconfiguration), or partially (partial reconfiguration) with a completely new bitstream. There are two basic approaches to partial reconfiguration.

1. Static Partial Reconfiguration: This involves reconfiguring the device when it is inactive, by putting the rest of the FPGA in shutdown mode during reconfiguration. Once the configuration is complete the FPGA is brought up once again. This is now a deprecated method of reconfiguration.

2. Partial Dynamic Reconfiguration (PDR): This involves reconfiguring a portion of the FPGA at run-time, without affecting the execution of other parts of the FPGA chip. Xilinx supports partial reconfiguration on the Virtex II, Virtex II Pro, and Virtex 4 FPGA families.

In order to understand PDR in greater detail the following terminology must be understood.

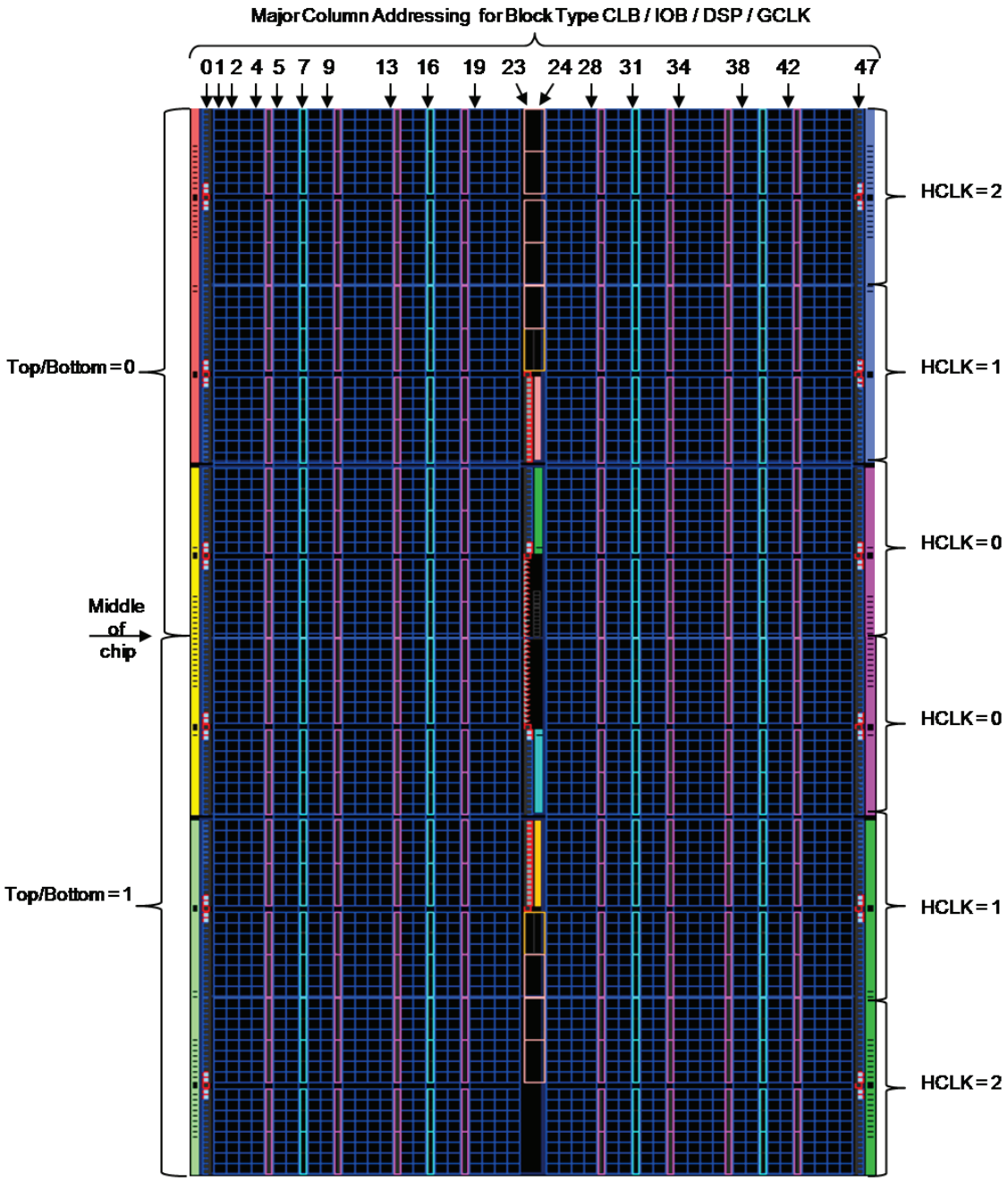

Fig. 2.6: Virtex 4 SX35 FPGA floorplan.

Table 2.1: Number of frames per column by resource type.

Column Type	Frames per column
CLB	22
IOB	30
DSP	21
CLK	2
BRAM Interconnect	20
BRAM Content	64

2.4.1 Partial Reconfiguration Region (PRR)

A portion on the FPGA chip which is marked for reconfiguration. PRRs are designated by the user at design time based on the size of the design being reconfigured.

2.4.2 Partial Reconfiguration Module (PRM)

This is the actual reconfigurable design that is placed at run-time inside a PRR. Any number of PRMS can reside within the same PRR.

2.4.3 Static Logic

The portion of an FPGA chip that does not change its logic during partial reconfiguration. This may typically also contain circuitry to control the partial reconfiguration process. It is also called as the Base Region. A typical PDR scenario is shown in Fig. 2.7 [11], where PRR A can be dynamically reconfigured as PRM A1, A2, or A3. In other words the PRR can function as either design A1, A2, or A3 depending on which PRM is configured into it.

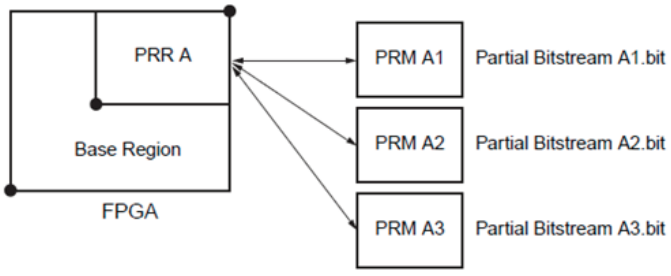

Fig. 2.7: Partial reconfiguration loaded with three Partial Reconfiguration Modules (PRMs).

2.4.4 Bus Macro (BM)

During partial reconfiguration one needs to ensure that the routing signals used for inter-module communication do not change when the module is reconfigured. This fixed routing bridge of communication is achieved by using bus macros.

Bus macros are pre-placed, pre-routed hard macros provided as part of Xilinx's EAPR tools, as a means of locking the routing between a PRM and the rest of chip, which includes the static logic as well as other PRMs. Bus macros have eight bits of data bandwidth and must straddle the boundary between the PRM and the base design. They cannot, however, straddle a DSP or BRAM column on the FPGA. Therefore bus macros are always placed manually during the chip floorplanning phase. There are several different types of bus macros, allowing designers to choose, as per the EAPR User Guide [11].

1. Signal direction: The physical placement of a PRR on the FPGA, and the orientation of its input and outputs signals for communication with static logic determine which type of bus macro will be used. Figure 2.8 depicts a typical FPGA floorplan with four different types of bus macros. PRR 0 in the figure being on the left side of the static logic, communicates via a Right-to-Left (R2L) bus macro for its inputs, and Left-to-Right (L2R) bus macro for its outputs. PRR1 uses a Top-to-Bottom (T2B) bus macro for its inputs, and Bottom-to-Top (B2T) bus macro for its outputs. Similarly, PRR 2 uses B2T bus macros for getting inputs, and T2B bus macros for output. PRR 3 uses L2R bus macros for input and R2L bus macros for output.

2. Physical width of the bus macro: Bus macros can be narrow or wide. The attributes wide and narrow refer to the physical width of the bus macro, and not the bandwidth. Narrow bus macros are two CLBs wide, while wide bus macros are four CLBs wide.

3. Synchronous versus asynchronous bus macros: Depending on whether signals passing through the bus macros are registered or not they can synchronous (registered input and output signals) or asynchronous (signals are not registered). Synchronous bus macros provide a superior timing performance, as long as the design can manage the additional latency.

2.4.5 Internal Configuration Access Port (ICAP)

The ICAP is the Xilinx device primitive that allows reading/writing of configuration data from/to the FPGA. The input and output ports for ICAP are shown in Fig. 2.9. All the configuration commands and the data being sent to the ICAP must be fed via the input port *I*. Configuration frames can be read out from the FPGA from the output port *O*. The *CE* and *WRITE* ports are used to control the read and write transactions. The ICAP can be configured in different working modes as follows:

1. ICAP Write Mode: ICAP is in write mode when the *CE* and *WRITE* pins are asserted low.

2. ICAP Read Mode: ICAP is in read mode when the *CE* pin is asserted low, and the *WRITE* pin is asserted high.

An important part of the ICAP command protocol is that the *CE* pin should remain high whenever the *WRITE* pin is changing states. If both the *CE* and *WRITE* pins are set high the ICAP goes into an abort state and configuration is halted.

Partial reconfiguration thus provides the flexibility to support a wide range of applications by time-sharing physical resources on-the-fly. This helps pack more functionality per chip to execute different designs with the same FPGA hardware. It also helps reduce the power consumption by eliminating idle circuit power. Configuration time is directly proportional to the size of the configuration bitstream. Partial reconfiguration allows you to make small modifications without having to reconfigure the entire device. By changing only portions of the bitstream - as opposed to reconfiguring the entire device - the total reconfiguration time is reduced. This is especially valuable where devices operate in a mission-critical environment that cannot be disrupted while some subsystems are being redefined [12]. Lastly, partial reconfiguration can also be used to remotely reconfigure hardware, which is particularly useful in application areas where in-the-field hardware upgrades are needed.

Several methods have been proposed to harness the partial reconfiguration capabilities in FPGAs, but have mostly turned out to be tedious and time-consuming to be performed

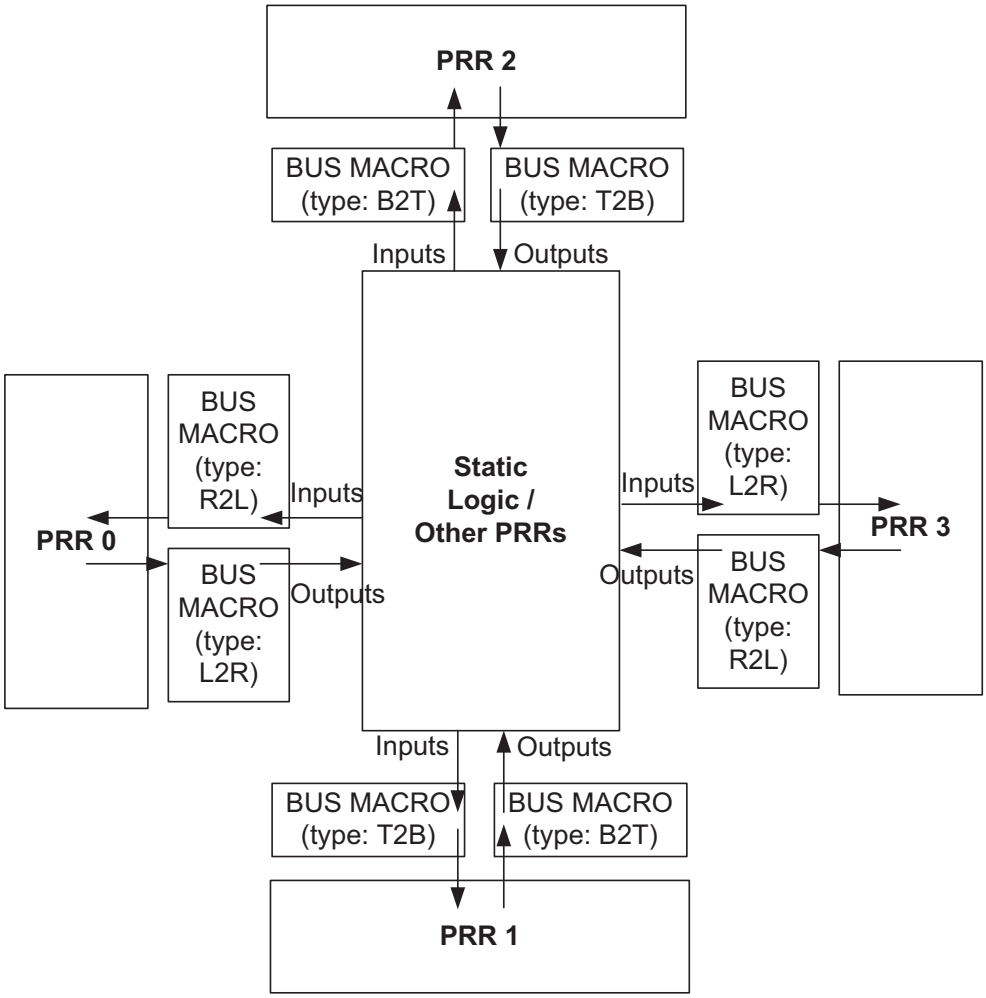

Fig. 2.8: Interaction between PRRs and static logic via bus macros.

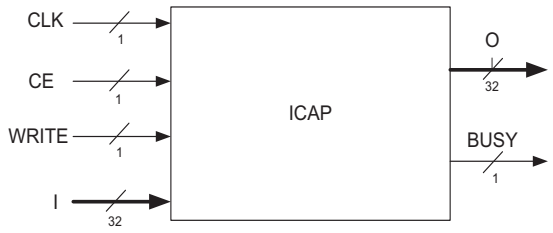

Fig. 2.9: ICAP primitive top-level schematic.

manually. Xilinx provides a design flow methodology called Early Access Partial Reconfiguration (EAPR) that helps in the partitioning of designs into static and dynamic regions, creation of PRMs, generation of partial and full bitstreams of a system, and the communication of PRRs with static logic via bus macros. The EAPR tool flow helps generate the netlists of the PRMs and the static logic to finally generate partial and full bitstreams for the design. This design flow methodology is shown in Fig. 2.10, and the phases involved are explained in the following sections.

2.4.6 HDL Design Description

During the initial phase, one needs to partition the design into (i) dynamically configurable parts which will form the PRMs, and (ii) static logic. Once the design has been partitioned, the PRMs and static logic are described in a HDL, like Verilog or Very-high-speed integrated circuit Hardware Description Language (VHDL).

2.4.7 Synthesis using Xilinx Integrated Synthesis Environment (ISE)

The designs are then synthesized using Xilinx ISE, a tool that compiles HDL code and generates the corresponding netlists for each design. For the reconfigurable modules it is important that the IOBs are removed, and the number of global clock buffers is set to zero. This is because PRMs typically cannot be physically constrained to route their clock and data input and output signals via fixed IOBs or global clocking resources. The designs need to be portable to any portion of the FPGA. The static logic can continue to use IOBs to communicate as its input and output ports.

2.4.8 System Design using Xilinx Platform Studio

Xilinx Platform Studio (XPS) provides a graphical user-interface to design an embedded System-on-Chip (SoC) with the static logic, PRMs, and other peripherals required to build a complete PDR solution. This includes the Xilinx Embedded Development Kit (EDK), which is a suite of tools and Intellectual Property (IP) cores that enables the designer to design a complete embedded processor system for implementation in a Xilinx FPGA device [13]. There is also an option to use a soft synthesizable processor core, called Microblaze [14] or a hard-core microprocessor like PowerPC to be used for running software designs as well. The MicroBlaze processor is a 32-bit Harvard Reduced Instruction Set Computer (RISC) architecture optimized for implementation on Xilinx FPGAs with separate 32-bit instruction and data buses running at full speed to execute programs and provide simultaneous access to data from both on-chip and external memory.

Figure 2.11 depicts a typical SoC for partial reconfiguration. The SoC design consists of the following components.

1. Different PRRs based on the number of PRMs we want executing simultaneously.
2. Array of bus macros to bridge communication between the PRRs and the static region.
3. Soft core (Microblaze) or embedded hard core microprocessor for executing the software part of the design.
4. Static Configuration Controller (SCC): The SCC is mainly responsible for performing the reconfiguration of PRRs. However its responsibilities may include much more, depending upon the system class in which it is used [15].
5. On-chip peripheral bus (OPB) that interconnects all the static region peripherals, processor, and bus macros.
6. On-chip memory/External Memory Controller (EMC): The system could either use on-chip memory or use an EMC to access off-chip memory, for storage of partial bitstreams.

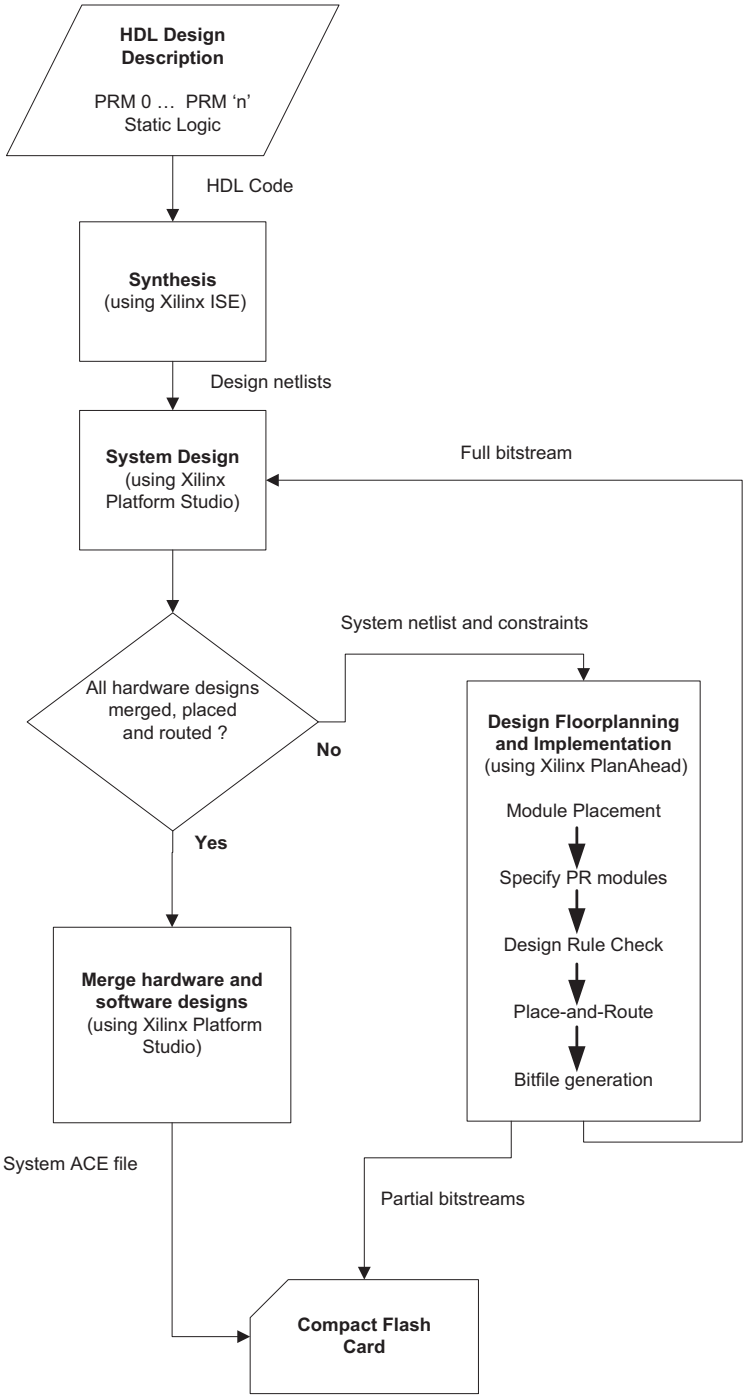

Fig. 2.10: EAPR design flow methodology.

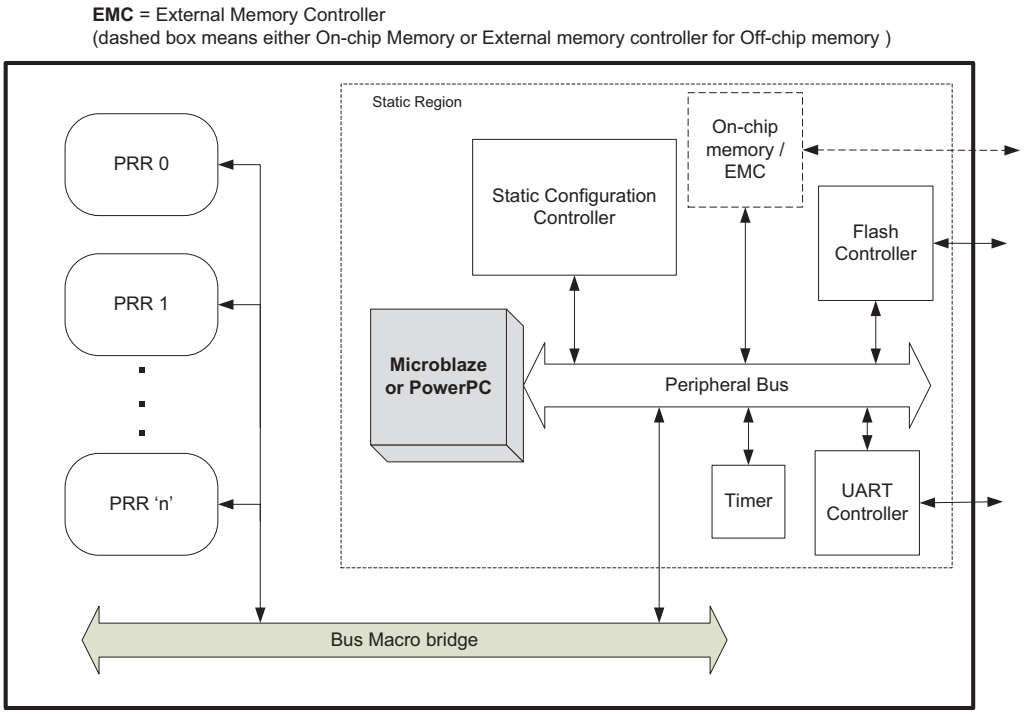

Fig. 2.11: Embedded SoC design for partial reconfiguration.

7. Any additional peripherals needed to communicate with external devices like the serial port, compact flash card, timer, etc.

Once the system has been designed in XPS, the system netlist and design constraints file are generated which are then ready for floorplanning.

2.4.9 Design Floorplanning and Implementation Using Xilinx PlanAhead

The system netlist and constraints specification from XPS is fed into Xilinx PlanAhead, a tool used to perform floorplanning and timing analysis for designs. The Xilinx PlanAhead software supports a floorplanning methodology that allows designers to constrain critical logic in order to ensure shorter interconnect lengths with lesser delays [16]. After the PRMs and the static logic have been placed in their respective regions on the chip, it is important to verify that the resource utilization of the region can accommodate each design. Xilinx PlanAhead can then be used to perform a Design Rule Check (DRC) on the design, so that the clocks, bus macros, static logic, and PRRs conform to the design constraints. Once the

design has been validated by the DRC, it can then be implemented to generate the partial bitstreams of the reconfigurable modules, and the full bitstream which contains both the static and reconfigurable portions of the design. The partial bitstreams are copied to the compact flash card, while the full bitstream is exported into XPS again.

2.4.10 Merging Hardware and Software Designs Using Xilinx Platform Studio

XPS allows users to specify a software design that will run on the on-chip microprocessor (MicroBlaze or PowerPC). After the user application and low-level device drivers and libraries have been compiled, the XPS tool merges the full bitstream from Xilinx PlanAhead with the compiler software to generate a final downloadable bitstream, called the System ACE file. The System ACE file is copied onto the compact flash card, and the card is plugged into the FPGA to bring up the design on the next power cycle.

2.5 Partial Bitstream Relocation (PBR)

Partial Bitstream Relocation involves moving partial bitstream data from one PRR to another PRR on the chip thus creating another copy of the source PRR's circuit. However partial bitstreams are closely tied to the physical region they are built for on the FPGA. This means, if there are N reconfigurable regions in a partially reconfigurable (PR) design, and M different reconfigurable modules that can be placed in these regions, we would have to generate $N \times M$ different partial bitstreams. Such a design constraint has two major drawbacks.

1. Wastage of (i) memory in storing a large number of partial bitstreams, when only N of these partial bitstreams are needed at any point in time, and (ii) power consumed in memory reads/write operations. Also the same circuit logic is stored redundantly as a different version for each PRR.

2. Partial bitstreams for most real-world circuits, like Discrete Wavelet Transform (DWT) and Discrete Cosine Transform (DCT), have sizes in the order of several kilobytes.

This will result in an unusually high reconfiguration time since one needs to read a new partial bitstream from memory, for each relocation to a different PRR.

Another alternative to this approach is to store only one copy of a circuit belonging to a single source PRR and then translate it accordingly for the destination PRR either in hardware or software. In order to better understand how this can be achieved a detailed understanding of the partial bitstream format and the configuration registers in Virtex-4 FPGAs is required. Figure 2.12 shows a top-level view of a partial bitstream for Virtex-4 FPGAs.

The bitstream is composed of the following components.

1. Proprietary Header: This contains file-specific information like the time and date of creation, the name of the source Native Circuit Description (NCD) file used to create the bitstream, and the FPGA chip family and model number.

2. Commands: The commands are used to communicate with the ICAP to set it up in write mode, and the number of frames to be written. This is then followed by the actual frame data. In order to read from the ICAP, commands set it up in read mode, and then start returning the frames read from the ICAP's data output port. Commands also help synchronize the ICAP before a read or write transaction, and desynchronize after the operation has completed. Finally commands can also trigger CRC calculations for frame data.

3. Frame Data: The frame data is the actual configuration data representing the values in different hardware resources (CLBs, IOBs, BRAMs, and DSPs) corresponding to the PRR being accessed via the ICAP.

In order to understand the internals of a partial bitstream, one needs to understand the various types of command packets, and configuration registers involved for sending instructions to the ICAP.

The Virtex-4 configuration logic consists of a packet processor, a set of registers, and global signals that are controlled by the configuration registers [17]. The packet processor

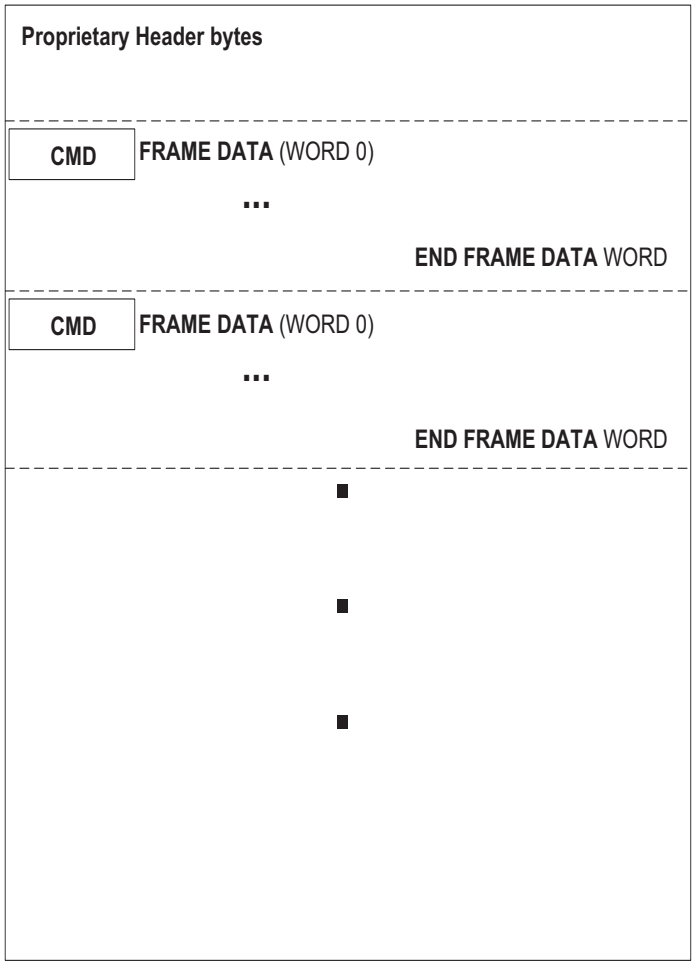

Fig. 2.12: Top-level partial bitstream format.

controls the flow of data from the configuration interface ICAP to the appropriate register. The registers control all other aspects of configuration.

The FPGA bitstream consists of two types of packets: Type 1 and Type 2. The Type 1 packet is used for register reads and writes. In Fig. 2.13, the Type 1 packet header contains the number of 32-bit words in the word count portion. Following the Type 1 header will be the Type 1 data equal to the number of words mentioned in the word count section of the header. Depending on the opcode (Fig. 2.14), the ICAP interprets the command as read, write or No-Operation (NOP).

The Type 2 packet, which must follow a Type 1 packet, is used to write long blocks. No address is presented here because it uses the previous Type 1 packet address. Figure

2.15 shows the Type 2 packet header format, with the lower 27 bits used for the word count. This is followed by Type 2 packet data, equal to the number of words mentioned in the word count section of the header.

All bitstream commands are executed by reading or writing to the configuration registers. Figure 2.16 summarizes the different configuration registers available in Virtex 4 FPGAs. These registers are explained in the Virtex-4 FPGA Configuration User Guide [17]. Of these the Frame Address Register (FAR), is particularly critical to PBR. The FAR has been discussed in Sec. 2.3.

Let us now consider a section of the partial bitstream and analyze the various commands present. Figure 2.17 depicts a section of hexadecimal data from a partial bitstream, with the dummy word (`FF FF FF FF`), sync word (`AA 99 55 66`), frame address (`00 00 41 80`), and the write command to the Frame Data Register Input (FDRI) (`30 00 40 29`).

The dummy word is used for periodically flushing the packet buffer before starting a read or write transaction with the ICAP. The sync word is used to mark a 32-bit word boundary for the ICAP to read frame data words from the partial bitstream. The frame address shown in Fig. 2.17 can be deciphered as follows.

Binary Sequence: 0000 0000 0000 0000 0100 0001 1000 0000

FAR Format: 000000000 **0 000 00001 00000110 000000**

The binary digits marked in bold, correspond to the bits mentioned in Fig. 2.5. Therefore the Top_Bottom Bit is set to `0`, meaning the frames address location is in the top half of the FPGA. The Block Type equals `000`, so the frames are of type CLB, DSP, IOB, or CLK. The Row Address is equal to `00001`, meaning this is HCLK row 1 (refer to Fig. 2.6 to find this HCLK row). The major column address is set to `00000110`, which refers to the

Header Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

Fig. 2.13: Type 1 packet header format.

sixth CLB column from the left of the FPGA. Finally the minor address is `000000`, which is the first minor frame number for that column.

The next command in the partial bitstream (`30 00 40 29`) in Fig. 2.17 can be interpreted as follows:

Binary Sequence: 0011 0000 0000 0000 0100 0000 0010 1001

Type 1 Packet Format: **001 10** 00000000**00010** 00 **00000101001**

Starting from the left, most bit patterns (marked in bold), and using the bit map in Fig. 2.13, we can see that the packet type is set to `001`, which means this is a Type 1 packet. The next two bits are set to `10`, so this specifies a write transaction. The lower five bits of the following 14-bit sequence are set to `00010`, which based on the information in Fig. 2.16 tells us that this is a command to the FDRI. Finally the lower 11 bits are set to `00000101001`, or 41 in decimal. Therefore, this command attempts to write 41 32-bit words or a single frame to the FDRI.

In this section we have reviewed the frame addressing and command sequences required to communicate with the ICAP for reading and writing frames. In the partial bitstream data from Fig. 2.17, by changing the frame address bits to a different location, we can write the same frame data to any other location on the FPGA as well. This is fundamental to the concept of PBR, because we can then essentially copy from a PRR to any other PRR without having to store it as a separate partial bitstream. This gives huge memory savings, because instead of storing N different partial bitstream versions of the same design, only one partial bitstream per design (PRM) needs to be stored.

Opcode	Function
00	NOP
01	Read
10	Write
11	Reserved

Fig. 2.14: Opcode format.

Header Type	Opcode	Word Count
[31:29]	[28:27]	[26:0]
010	RR	xxxxxxxxxxxxxxxxxxxxxxxxxxx

Fig. 2.15: Type 2 packet header format.

Reg. Name	Read/Write	Address	Description
CRC	Read/Write	00000	CRC register
FAR	Read/Write	00001	Frame Address Register
FDRI	Write	00010	Frame Data Register, Input (write configuration data)
FDRO	Read	00011	Frame Data Register, Output register (read configuration data)
CMD	Read/Write	00100	Command Register
CTL	Read/Write	00101	Control Register

Fig. 2.16: Configuration registers.

2.6 Literature Review

2.6.1 Partial Dynamic Reconfiguration

PDR techniques stemmed from the need to reduce reconfiguration time in reconfigurable computing. Mesquita et al. [18] explain the proliferation of PDR in the Virtex family of FPGAs with the onset of larger and more complex SoC architectures in recent years. PaDReH [19] is a design tool that takes a system's design specifications in Sys-

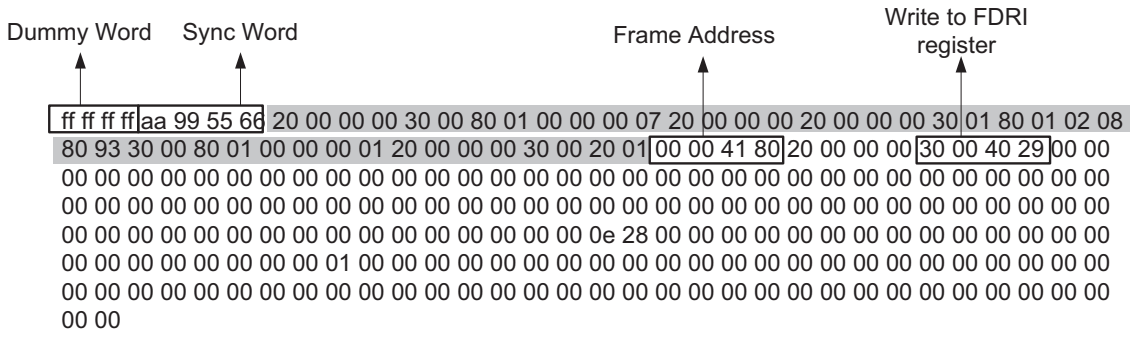

Fig. 2.17: Partial bitstream showing commands and frame data.

temC [20], converts it to Register Transfer Level (RTL) code in terms of Verilog or VHDL, performs functional validation of the HDL design, followed by partitioning and scheduling of HDL designs, and followed by reconfigurable infrastructure generation steps to implement a Dynamically and Partially Reconfigurable System (DRS) on the FPGA. Caronte [21] uses Xilinx EDK (embedded development kit) to generate Blackboxes, which are regions on the FPGA that contain a architecture-dependent interface built using bus macros and a processing element which can be reconfigured at run-time using ICAP. JPG [22] is a tool which generates partial bitstreams using the JBits Java API available for certain Virtex families of Xilinx FPGAs. The JBits API [23] helps provide off-chip software manipulation of partial bitstreams, which are then downloaded onto the FPGA for reconfiguration. FIGARO [24] provides a partial reconfiguration tool for Atmel FPGAs.

Sedcole et al. [25] propose two methodologies to dynamic reconfiguration: direct dynamic configuration and merge dynamic reconfiguration. Direct dynamic reconfiguration configures the entire FPGA's configuration memory with a design and then uses partial bitstreams to reconfigure sections of the FPGA. This method was not found to be resource efficient and suffers from timing closure issues for large modules. Merge dynamic reconfiguration is a novel technique of reconfiguration where a bitstream is read frame-by-frame and an exclusive-OR (XOR) operation is performed with another candidate bitstream, essentially merging them together, before being written back to the reconfigurable region. This enables two designs to co-exist within the same PRR since, one can always XOR out one of the designs. The major concern here is that the reconfiguration is high for a read-modify-write model of reconfiguration.

Diessel and Milne [26] have developed an FPGA compiler that takes as its input, abstract system descriptions written in Circal process algebra [27], to generate the corresponding FPGA reconfigurable logic for a specified design. Gericota et al. [28] provide a method of dynamically relocating CLBs and routing resources using JBits [23] based on what functionality needs to be reconfigured into the FPGA. Tan and DeMara [29] present a physical resource management strategy for minimizing reconfiguration overhead by plac-

ing the logic elements in the least number of slice columns as possible, with the added complexity of managing the routing between the newly placed logic elements. Singhal and Bozorgzadeh [30] suggest another technique to reduce the reconfiguration time by not reconfiguring the common components and only reconfiguring the difference in the source and destinations PRMs. However the differences in the routing and placement of the common blocks always limit the total reconfiguration time saved. The benefits of PDR are limited by the reconfiguration time involved, which includes the use of co-processors to control reconfigurability.

2.6.2 Partial Bitstream Relocation

PBR techniques can be categorized based on the following attributes:

1. Location of bitstream storage (off-chip or on-chip),
2. Type of ICAP wrapper (Xilinx pre-defined Hardware ICAP (HWICAP) or custom ICAP wrapper),
3. Relocation logic (hardware or software),
4. Type of processor core (soft synthesizable core or hard core microprocessor),
5. Mode of bitstream relocation (active or passive),
6. Source and destination PRR resource sets (identical or non-identical).

Early bitstream relocation techniques like PARBIT [31] involved using an off-chip processor to generate partial bitstreams from a given input bitstream for different PRRs, which were then used for bitstream relocation. pBITPOS [32] is an improvement over techniques like PARBIT where support for relocation on Virtex-II FPGAs and bitstream manipulation for BRAMs and multiplier columns was added. Many techniques use the off-chip flash memory or on-chip Block RAMs (BRAMs) to store the partial bitstream. The partial bitstream is then typically transformed for the target PRR, using either a hardware-based implementation or relocation software running on an on-chip processor. Montminy et al. [33]

use a software-based relocation technique implemented on a hard-core microprocessor to relocate partial bitstreams for fault-tolerant designs, using the Xilinx IP core HWICAP. Their implementation is limited to the Virtex-II Pro family of FPGAs and performs CRC calculations on the bitstream in software as well.

REPLICA [2] is a hardware based relocation filter that sits between the on-chip memory and target region, thus transforming the addressing in the bitstream on the fly. It, however, does not support BRAM and multiplier columns to be relocated. REPLICA2Pro [3] is an improved version of REPLICA that supports relocation of BRAM and multiplier columns, as well as other Xilinx FPGA families like Virtex II and Virtex II Pro. However, they use off-chip memory to store the partial bitstream and are restricted to relocating PRRs that are at most one frame tall. They also have a significant control data overhead due to column-wise partial bitstream relocation. Corbetta et al. [4] propose another hardware based relocation filter Bit Relocation Filter (BiRF) that can target Virtex 4 and 5 series of the Xilinx FPGAs in addition to Virtex II. Carver et al. [5] proposed a software-based relocation technique that significantly reduces the design time, and compile time for partial bitstreams. Becker et al. [1] proposed a novel technique to relocate between non-identical regions on the FPGA by replacing the non-identical frames between the source and target PRRs with pad frames and necessary routing logic. They achieve this by writing a software driver around the Xilinx's HWICAP IP core. However, a major drawback of their technique is the storage of the partial bitstream on off-chip memory which slows down relocation time. All of the previous discussed techniques suffer from performance bottlenecks either attributed to the use of software-based relocation techniques or hardware-based relocation filters that rely on off-chip memory for partial bitstream storage. The relocation time further goes up when CRC calculations need to be performed after the bitstream has been relocated.

Frame Data Relocation (FDR) is a recent advancement in PBR where the bitstream relocation is performed on an active PRR by reading its configuration on-the-fly one frame at a time into another PRR. Sudarsanam et al. [6] were the first to innovate with this idea, which had negligible on-chip BRAM usage, and 153x performance increase in relocation

time as compared to contemporary relocation techniques like BiRF [4].

Chapter 3

Multi-row PRR-PRR Relocation

In the following section (Sec. 3.1), we introduce single row PRR-PRR relocation, after which the need for a multi-row PRR-PRR relocation model is explored (Sec. 3.2), which is followed by a discussion of the proposed multi-row PRR-PRR relocation algorithm (Sect. 3.3).

3.1 Single-row PRR-PRR Relocation

PRR-PRR relocation [6] is a bitstream relocation technique where a source PRR's frame data is relocated to a destination PRR, while the source PRR's circuit is still running. This is essentially the active bitstream relocation technique discussed in Sec. 2.6.2. Figure 3.1 shows a basic PRR-PRR relocation setup with three PRRs (*PRR 0*, *PRR 1*, and *PRR 2*) and a static region. The relocation is carried out using the Accelerated Relocation Circuit (ARC) methodology described by Sudarsanam et al. [6].

Let us now consider the relocation path between *PRR 0* and *PRR 1* for example. Notice that both the PRRs are symmetric, which is a prerequisite for PRR-PRR relocation using ARC. During the PRR-PRR relocation between *PRR 0* and *PRR 1*, each frame of *PRR 0* is copied and stored in a temporary BRAM buffer by the ARC and then copied back to *PRR 1*. These observations from Fig. 3.1 can be summarized as follows.

1. Accelerated Relocation Circuit (ARC): This is similar to a Static Configuration Controller (SCC) in functionality as explained in Sec. 2.4. When instructed to relocate from one PRR to another, it copies active bitstream information from PRR to another frame by frame. As discussed in Sec. 2.6.2, doing this results in huge memory savings because the ARC needs to only store one frame, which is essentially one BRAM

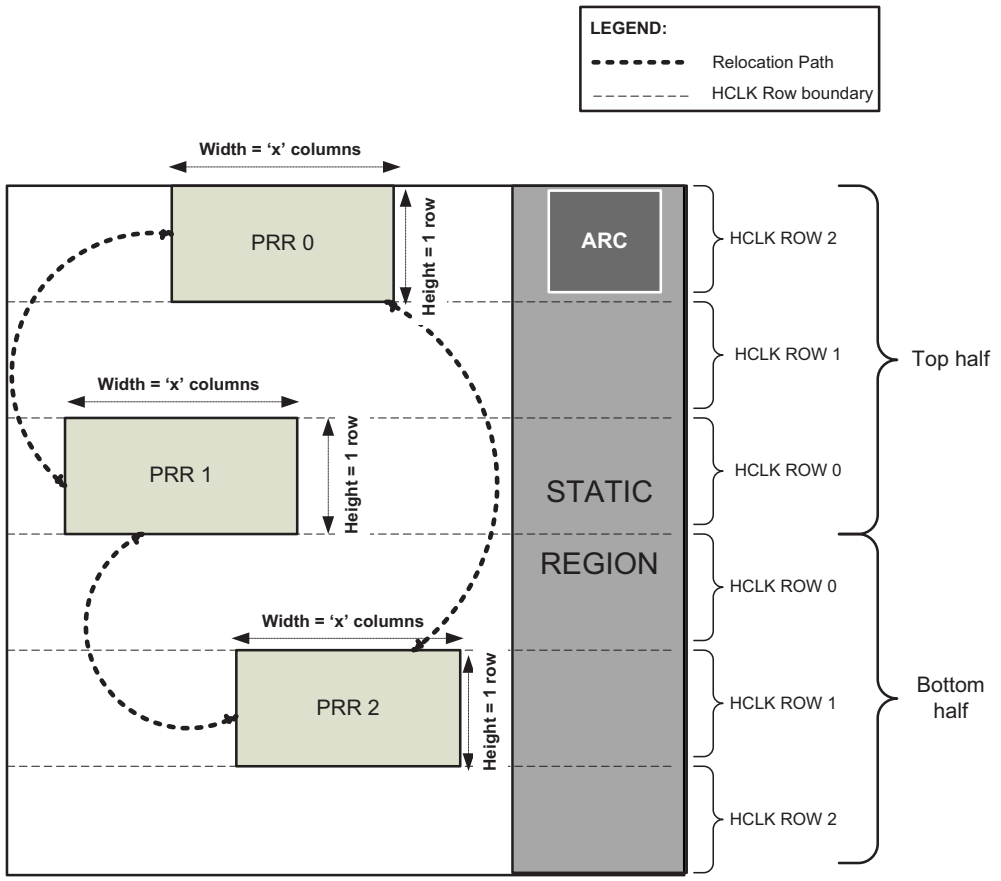

Fig. 3.1: Basic PRR-PRR relocation support on a Xilinx Virtex-4-SX35 FPGA.

resource. The different relocation possibilities between all three PRRs are shown by solid dashed lines in Fig. 3.1.

2. Xilinx Virtex 4 SX35 FPGA Floorplan: The above PRR-PRR relocation scenario is for a Virtex 4 SX35 FPGA. This versatile FPGA architecture has its chip resources organized as six HCLK rows, equally distributed in the top and bottom half of the FPGA. A detailed description of Virtex 4 SX35 can found in Sec. 2.3.

3. Dimensions of PRR: ARC requires that all PRRs be of similar dimensions. This is because the ARC cannot relocate from one PRR to another PRR with different number of CLB, DSP, or BRAM columns. Each PRR's width in Fig. 3.1 is equal to x columns, and the height is equal to one HCLK row. The HCLK rows are demarcated by dashed lines in Fig. 3.1.

From the previous discussion it is evident that the ARC has so far supported single row based PRRs. One of the contributions of my thesis is to add support for multi-row PRR-PRR relocation by extending ARC's capabilities, which is a common necessity in large designs.

3.2 Need for Multi-row PRR-PRR Relocation

Multi-row PRR-PRR relocation helps provide a wider range of design choices based on different width and height specifications available on the FPGA floorplan. The multi-row PRR-PRR relocation technique can be better visualized from Fig. 3.2.

Multi-row PRR-PRR relocation has the following advantages.

1. Support for larger multi-row designs: PRRs can now occupy portions that are x (≥ 1) columns wide and y (≥ 1) rows in height, where x can be the entire width of the chip and y the maximum number of rows available. Thus a design is no longer constrained to fit within a single HCLK row.

2. Added flexibility in designing PRRs: For large PRRs multi-row support can provide the added flexibility to fit the same design in different HCLK rows and columns. For example in Fig. 3.2, *PRR1* could occupy half the number of columns, and be expanded length-wise to occupy double the number of rows. Thus one can better utilize the space on the FPGA, instead of constraining PRR design to a single row. The designer will however need to ensure that PRRs are identical.

Thus not only are PRRs larger than 1 HCLK row, but also PRR design can be tailored for more efficient placement and routing.

3.3 Multi-row PRR-PRR Relocation Algorithm

The proposed multi-row PRR-PRR relocation algorithm, shown in Fig. 3.4, takes the source and destination region addresses as its inputs.

Each region address provides: (i) the start and end row addressing, (ii) the start and end column addressing, and (iii) which portion of the FPGA do the rows start and end in,

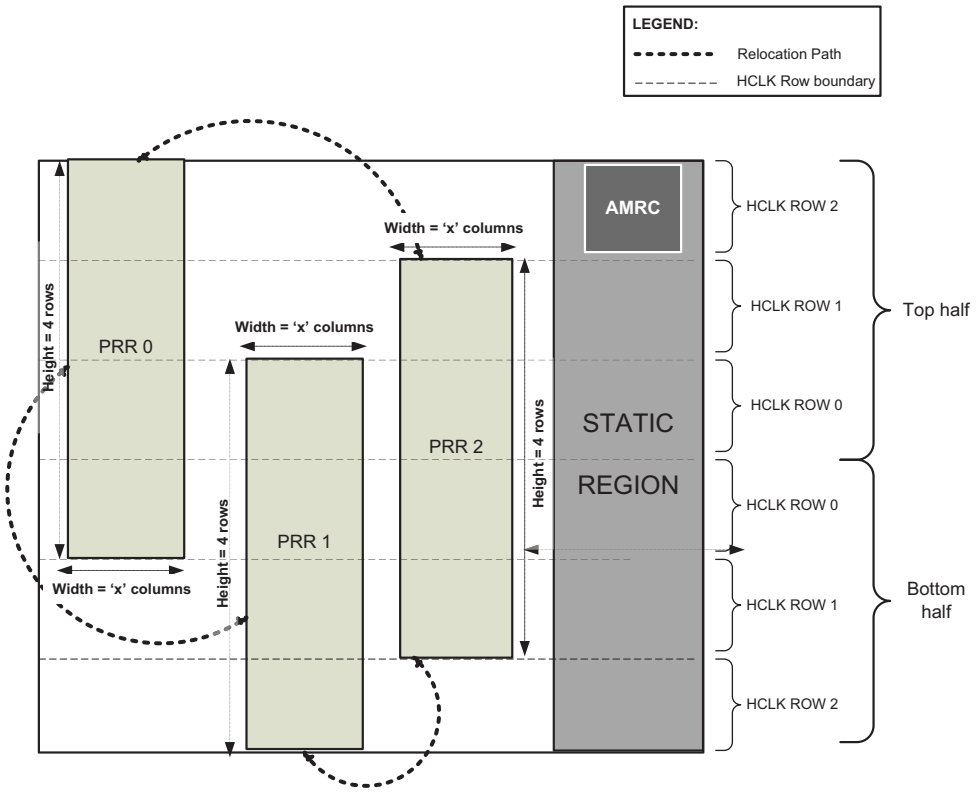

Fig. 3.2: Proposed multi-row PRR-PRR relocation support on a Xilinx Virtex-4 SX35 FPGA.

i.e., whether the PRR is in the top half, bottom half, or spans across the middle of the chip. Based on this information, the number of rows (*FindRowCount* routine in Fig. 3.3), and columns to be addressed (*NumberofMajorColumns*) are calculated. After this the block type of each column is found, and by iterating through each minor column (*step 7.A.iv* in Fig. 3.4), a source and destination frame address is generated every time. This process is repeated for every column in the current row. After the current row has been relocated, the next HCLK row is processed in a similar fashion (routines *FindNextHCLKRow* and *FindNextTopBottomBit* shown in Fig. 3.3). The relocation is completed after iterating through each column of each row in the source and destination PRRs. Another important routine in Fig. 3.3 is BitReversal, which generates a mirror image of the bits inside each frame if the source and destination frame addresses are on opposite sides of the FPGA.

FindNextTopBottomBit (CurrentTopBottomBit, CurrentRow)

1. If CurrentTopBottomBit = 0 and CurrentRow = 0
   Return 1
   Else
   Return CurrentTopBottomBit

FindNextHCLKRow (CurrentRow, TopBottomBit)

1. If TopBottomBit = 0
   A. If CurrentRow > 0 (*If row is in the top half, decrement row address*)
   Return (CurrentRow – 1)
   Else (*crossed over middle of the chip*)
   Return 0
   Else (*If row is in the bottom half, increment row address*)
   Return (CurrentRow + 1)

FindRowCount (SrcRowAddrStart, SrcRowAddrEnd)

1. If SrcRowAddrStart = SrcRowAddrEnd (*Source and Destination row is on the same side of chip*)
   A. If SrcRowAddrStart = 0 (*Source and destination row is in the top half*)
   RowCount = SrcRowAddrStart – SrcRowAddrEnd + 1
   Else (*Source and destination row is in the bottom half*)
   RowCount = SrcRowAddrEnd – SrcRowAddrStart + 1
   Else
   RowCount = SrcRowAddrStart + SrcRowAddrEnd + 2
2. Return RowCount

Relocate (FarSrc, FARDest, TopBottomBitFARSrc, TopBottomBitFARDest)

1. FrameData = ReadFrame(FARSrc)
2. FrameDataModified = **BitReversal**(FrameData, TopBottomBitFARSrc, TopBottomBitFARDest)
3. WriteFrame(FARDest)

BitReversal (FrameData, TopBottomBitFARSrc, TopBottomBitFARDest)

1. If TopBottomBitFARSrc = TopBottomBitFARDest (*Source and Destination frame is on same side of chip*)
   FrameDataModified = FrameData
   Else
   FrameDataModified = **MirrorImage**(FrameData)
2. Return FrameDataModified

MirrorImage (FrameData)

1. FrameDataModified = Reverse the bits in each word except the middle word in the frame
2. Return FrameDataModified

Fig. 3.3: Multi-row PRR-PRR relocation algorithm (subroutines).

Multi-row_PRR-PRR_relocation (SrcRA, DestRA)

BlockTypeList[] = {IO, CLB, CLB, CLB, CLB, BRAM, CLB, CLB, DSP48, …}

1. Read SrcRowAddrStart, SrcRowAddrEnd, TopBottomBitSrc from SrcRA
2. Read DestRowAddrStart, DestrRowAddrEnd, TopBottomBitDest from DestRA
3. RowCount = **FindRowCount**(SrcRowAddrStart, SrcRowAddrEnd)
4. Read NumberofMajorColumns from SrcPRR
5. CurrentRowSrc = SrcRowAddrStart
6. CurrentRowDest = DestRowAddrStart
7. For i = 1 to RowCount
   - A. For j = 1 to NumberofMajorColumns
     - i. Identify MajorColumnAddrSrc and MajorColumnAddrDest
     - ii. BlockType = BlockTypeList[MajorColumnAddrSrc]
     - iii. Identify NumberofMinorColumns based on BlockType
     - iv. For MinorColumnAddr = 0 to (NumberofMinorColumns – 1)
       - [1] Generate FARSrc
       - [2] TopBottomBitFARSrc = Bit index 22 of FARSrc
       - [3] Generate FARDest
       - [4] TopBottomBitFARDest = Bit index 22 of FARDest
       - [5] **Relocate** (FarSrc, FARDest, TopBottomBitFARSrc, TopBottomBitFARDest)
     - v. End For
   - B. End For
   - C. NextTopBottomBitSrc = **FindNextTopBottomBit**(TopBottomBitSrc, CurrentRowSrc)
   - D. NextTopBottomBitDest = **FindNextTopBottomBit**(TopBottomBitDest, CurrentRowDest)
   - E. CurrentRowSrc = **FindNextHCLKRow**(CurrentRowSrc, TopBottomBitSrc)
   - F. CurrentRowDest = **FindNextHCLKRow**(CurrentRowDest, TopBottomBitDest)
   - G. TopBottomBitSrc = NextTopBottomBitSrc
   - H. TopBottomBitDest = NextTopBottomBitDest
8. End For

Fig. 3.4: Proposed multi-row PRR-PRR relocation support on a Xilinx Virtex-4 SX35 FPGA.

Chapter 4

Memory-based Frame Data Reconfiguration (M-FDR)

M-FDR is a novel reconfiguration technique proposed in this thesis and is a potentially faster and a more flexible alternative as compared to the current state-of-the-art PBR methods like PRR-PRR relocation using ARC [6] and BiRF [4]. Section 4.1 discusses the proposed M-FDR design including its advantages, followed by a detailed explanation of the performance prediction model in Sec. 4.2 to help quantify the performance speedup achieved through M-FDR.

4.1 M-FDR Design

M-FDR design consists of the following components.

1. On-Chip Memory: The on-chip memory is an array of BRAMs used to store an entire PRR's frame data. On-chip frame data storage not only helps in faster access during reconfiguration, but is also more space efficient than storing multiple partial bitstreams for designs.

2. Accelerated Memory-based Reconfiguration Circuit (AMRC): The AMRC is the hardware circuit that drives the M-FDR design for each M-FDR operation. There are two types of M-FDR operations supported by the AMRC.

   (i) Frame Copy: The source for this operation is a particular PRR, and the destination is on-chip memory. This operation copies frames from the source PRR into memory one frame at a time. Therefore, we can dump an entire PRR's frame data into memory.

   (ii) Frame Configure: The source for this operation is the on-chip memory and the destination is a particular PRR. This operation configures a PRR with the

frame data stored in memory, typically in blocks of frames since we have the entire PRR's frame data residing in memory.

We will look into the hardware architecture internals of AMRC in greater detail in Chapter 5. Figure 4.1 shows a visualization of using M-FDR to relocate from *PRR 0* to *PRR 1* and *PRR 2*.

M-FDR has benefits both in terms of performance and space utilization. In terms of performance, M-FDR is better than PRR-PRR relocation [6] because of faster relocation times achieved by writing frames in bulk when compared to writing one frame at a time. Bulk frame writes during frame configuration have significantly low reconfiguration overhead because they only incur frame overhead (synchronization, write commands, and desynchronization) for groups of 20 to 44 frames. The space utilization is also much higher than other PBR techniques that rely on memory like BiRF [4] because the frame data is smaller in size as compared to the corresponding partial bitstream.

4.2 Performance Estimation Model

In order to analyze the performance speedup in M-FDR compared to PRR-PRR relocation, an analytical model is presented that identifies the different factors contributing to reconfiguration time. The unit of time is clock cycles, and the frame word length is 32 bits. Each PR design occupies a certain number of frames (nFrames) on the FPGA floorplan based on (i) design size and resource usage, and (ii) result of the place-and-route phase during design floorplanning using EAPR tools (refer to the EAPR design flow in Fig. 2.10). Table 4.1 lists all the parameters that will be used in this performance prediction model. The overall time to perform M-FDR is given by eq. (4.1). The value of $T_{Configure}$ is determined by eq. (4.2) and eq. (4.3).

There are three steps involved in copying frame data from the ICAP. The first step comprises of a set of synchronization commands that clear the internal packet buffer and set up ICAP in read mode. The second step is the process of reading the Frame Data (FD) from the ICAP and storing it in on-chip memory. The third step comprises of a set of

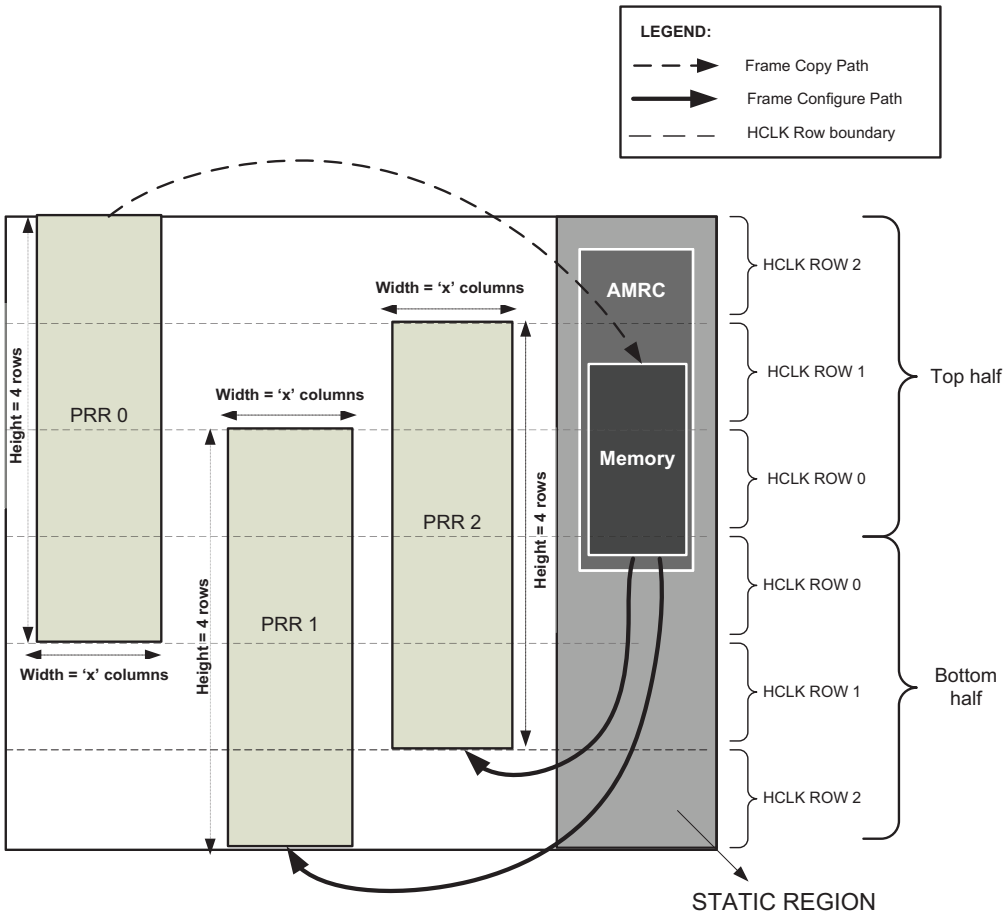

Fig. 4.1: Using M-FDR to relocate from *PRR 0* to *PRR 1* and *PRR 2*.

de-synchronization commands to flush the internal packet buffer and terminate the reading process. This protocol is followed until all frames in the source PRR have been copied to memory. Time taken to copy each frame's data is given by the sum of parameters *4*, *5*, *6*, *8*, and *9* from Table 4.1.

$$T_{Overall} = (nFrames \times T_{Copy}) + T_{Configure} + T_{BitReversal} \tag{4.1}$$

$$T_{Configure} = (T_{WriteOverhead} \times Factor_{WriteOverhead}) + (T^{FD}_{writeICAP} \times nFrames) \tag{4.2}$$

Table 4.1: Parameters used in the proposed performance model.

Parameter	Name	Description
1	T_{Copy}	Time taken to copy frame data from ICAP to memory
2	$T_{Configure}$	Time taken to configure FD to ICAP
3	$T_{BitReversal}$	Time taken to reverse bits in case the frame is relocated to the opposite half of the FPGA
4	$T_{gen}^{syncRdCmds}$	Time taken to generate set-up commands, and store them in a buffer
5	$T_{writeICAP}^{syncRdCmds}$	Time taken to write set-up commands to ICAP
6	$T_{readICAP}^{FD}$	Time taken to read FD from ICAP
7	$T_{writeICAP}^{FD}$	Time taken to write FD to ICAP
8	$T_{gen}^{desyncCmds}$	Time taken to generate de-synchronization commands, and store them in a buffer
9	$T_{writeICAP}^{desyncCmds}$	Time taken to write de-synchronization commands to ICAP
10	$T_{WriteOverhead}$	Total overhead time involved in one bulk write of FD to ICAP
11	$Factor_{WriteOverhead}$	Overhead factor to calculate number of times bulk FD is written to ICAP
12	*isBramUsed*	Value set to 1 if there are BRAM columns in the destination PRR, else set to 0
13	*nBramFrames*	Total number of BRAM frames in the destination PRR
14	*nBramColumns*	Total number of BRAM columns in the destination PRR
15	*nOddColumns*	Total number of odd columns between consecutive BRAM or CLK columns in the destination PRR

$$
\begin{aligned}
Factor_{WriteOverhead} \quad = \quad & \left\lceil \frac{nFrames - (isBramUsed \times nBramFrames)}{44} \right\rceil \\
& + (isBramUsed \times nBramColumns) \\
& + (\lceil (0.3 \times nFrames) \rceil) \\
& + \left\lfloor \frac{nOddColumns}{2} \right\rfloor)
\end{aligned} \tag{4.3}
$$

Frame configuration involves moving the frame data from memory to another destination PRR. This involves four basic steps and is different from the frame copy operation in the way frame data is written to the ICAP. The first step is to identify pairs of CLB/DSP columns, or BRAM columns in the source PRR which forms chunks of frames. The second step comprises of a set of synchronization commands that clear the internal packet buffer and set up the ICAP in write mode. In the third step, each chunk of frames is sent using a bulk write via the ICAP. When writing frames in bulk write mode, only the starting frame's destination address is needed after which the ICAP auto-increments the addresses for every consecutive frame. The fourth step comprises of a set of de-synchronization commands to flush the internal packet buffer and terminate the writing process. Frame overhead time ($T_{WriteOverhead}$) for each bulk write via the ICAP is given by the sum of parameters *4*,

5, *8*, and *9* from Table 4.1. An important observation to make here is that according to eq. (4.1), the time taken to write one frame ($T^{FD}_{writeICAP}$) is simply multiplied by the total number of frames in the source PRR, and that the frame overhead time ($T_{WriteOverhead}$) does not scale linearly with the number of frames.

All of the parameters *4* through *9* in Table 4.1 can be generically termed as T^{α}_{gen}, $T^{\beta}_{writeICAP}$, and $T^{\gamma}_{readICAP}$, where α, β, and γ are the number of words in the frame data being processed. Each of these parameters is computed as a sum of the FPGA's startup latency before writing to ICAP starts ($T_{StartupLatencyICAP}$) and the time taken to write x words to the ICAP ($T_{write}(x)$) where x can be α, β, or γ.

Chapter 5

AMRC Implementation

In this section, the implementation details of the proposed M-FDR architecture are presented. Section 5.1 discusses the M-FDR hardware (AMRC), followed by a performance analysis of AMRC in Sec. 5.2.

5.1 M-FDR Hardware - Accelerated Memory Reconfiguration Circuit (AMRC)

The AMRC is the hardware implementation of the M-FDR design technique of bit-stream relocation. Figure 5.1 shows the general system architecture with the AMRC. The AMRC consists of the following five components: (1) AMRC Controller, (2) FAR Generator, (3) Frame Data Buffer (FDB), (4) Relocator, and (5) ICAP wrapper.

The AMRC Controller uniquely identifies each source and destination PRR using region addresses, called the Source Region Address (*SRA*) and Destination Region Address (*DRA*) respectively. The 24-bit region addressing format is shown in Fig. 5.2. The Row_{start} and Row_{end} refer to the HCLK rows occupied by the PRR. Similarly the $Column_{start}$ and $Column_{end}$ bits refer to the major columns occupied by the PRR. The $Top/Bottom_{start}$ and $Top/Bottom_{end}$ bits are used to indicate whether the PRR sits in either half of the FPGA, or extends across both halves.

A top-level block diagram of the AMRC is shown in Fig. 5.3. AMRC can be driven via software or hardware state machine to input the *SRA* and *DRA*. The AMRC controller can be sent commands by setting the AMRC mode accordingly. The following sub-sections discuss the five hardware blocks within the AMRC.

5.1.1 AMRC Controller

The AMRC controller drives all the other components of the system based on the

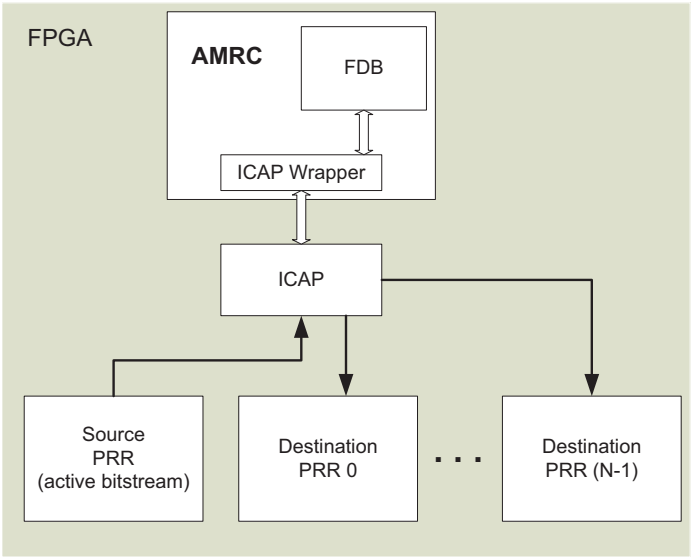

Fig. 5.1: System architecture overview with AMRC.

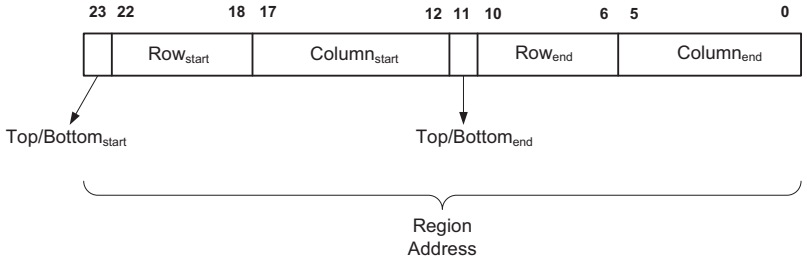

Fig. 5.2: AMRC PRR addressing format.

AMRC mode. Based on the current AMRC mode the FDB, Relocator and FAR Generator modules are appropriately configured by the AMRC controller. The different AMRC modes are as follows:

1. Reset Mode: In this mode, all components are held in reset. This is typically useful in clearing the state of all modules in the system, which is especially useful when changing between the other AMRC modes.

2. Relocate Mode: This mode is used to perform PRR-PRR relocation, i.e., the active bitstream in one PRR is transferred to another PRR frame by frame. The AMRC controller configures the Relocator module to perform PRR-PRR relocation based on

the FAR_{Src} and FAR_{Dest} outputs from the FAR Generator module. The FDB is in reset during this mode.

3. Copy Mode: In this mode, the AMRC controller copies frames from a particular source PRR into on-chip memory, the Frame Data Buffer (FDB). The AMRC controller configures the Relocator module to perform frame copy from the FAR_{Src} generated by the FAR Generator module, to the $FDB_{address}$ generated by the FDB module. The FDB_{mode} is set to write, while the $Relocator_{mode}$ is set to copy.

4. Configure Mode: In this mode, the AMRC controller configures frames from the FDB into a particular destination PRR. The AMRC controller configures the Relocator module to perform frame configuration from the $FDB_{address}$ generated by the FDB module to the $FAR_{Configure}$ which is the destination PRR address in this mode. The FDB_{mode} is set to read, while the $Relocator_{mode}$ is set to configure.

The AMRC controller is also responsible for configuring the source and destination address multiplexers that feed the Relocator (refer to Fig. 5.3 for *SrcMUX*, *DestMUX 1* and *DestMUX 2*). The AMRC controller can therefore be used for configuring the AMRC on-the-fly for PRR-PRR relocation, frame copy, and frame configure operations.

5.1.2 FAR Generator

The FAR Generator is tasked with generating source and destination frame addresses for PRR-PRR relocation and frame copy operations. Figure 5.4 highlights the process of generating frame addresses from a region address. It takes the *SRA* and *DRA* as its inputs, which are then decoded to the corresponding frame addresses for the source and destination PRRs. FAR Generator has two instances, one to decode the *SRA* and the other to decode the *DRA*. Each time a new frame address pair (source and destination) is generated, the FAR Generator sends a FAR_{done} signal to the AMRC controller. When the AMRC controller has finished a relocation or frame copy operation between those frame addresses, it sends an acknowledgement via the FAR_{go} signal. After the FAR Generator receives the acknowledgement from the AMRC Controller, it continues to generate the next

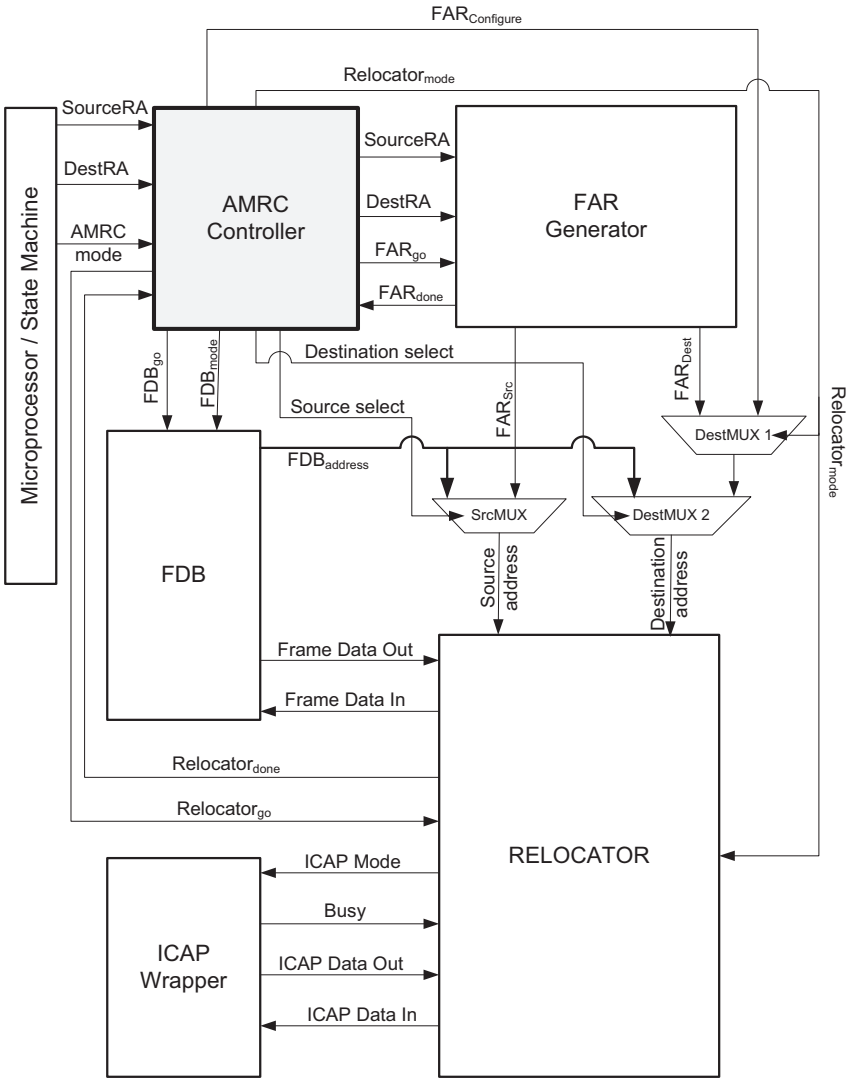

Fig. 5.3: AMRC block diagram.

frame address pair. This protocol is followed until all the frame addresses between the source PRR and destination PRR have been generated.

5.1.3 Frame Data Buffer (FDB)

The FDB stores the entire frame data that belongs to a source PRR. The FDB can be configured in read or write mode. During read mode the FDB gives out as output the frame data stored at the input address ($FDB_{address}$). In write mode the FDB stores the frame data at the given input address ($FDB_{address}$). An important feature in the FDB is that

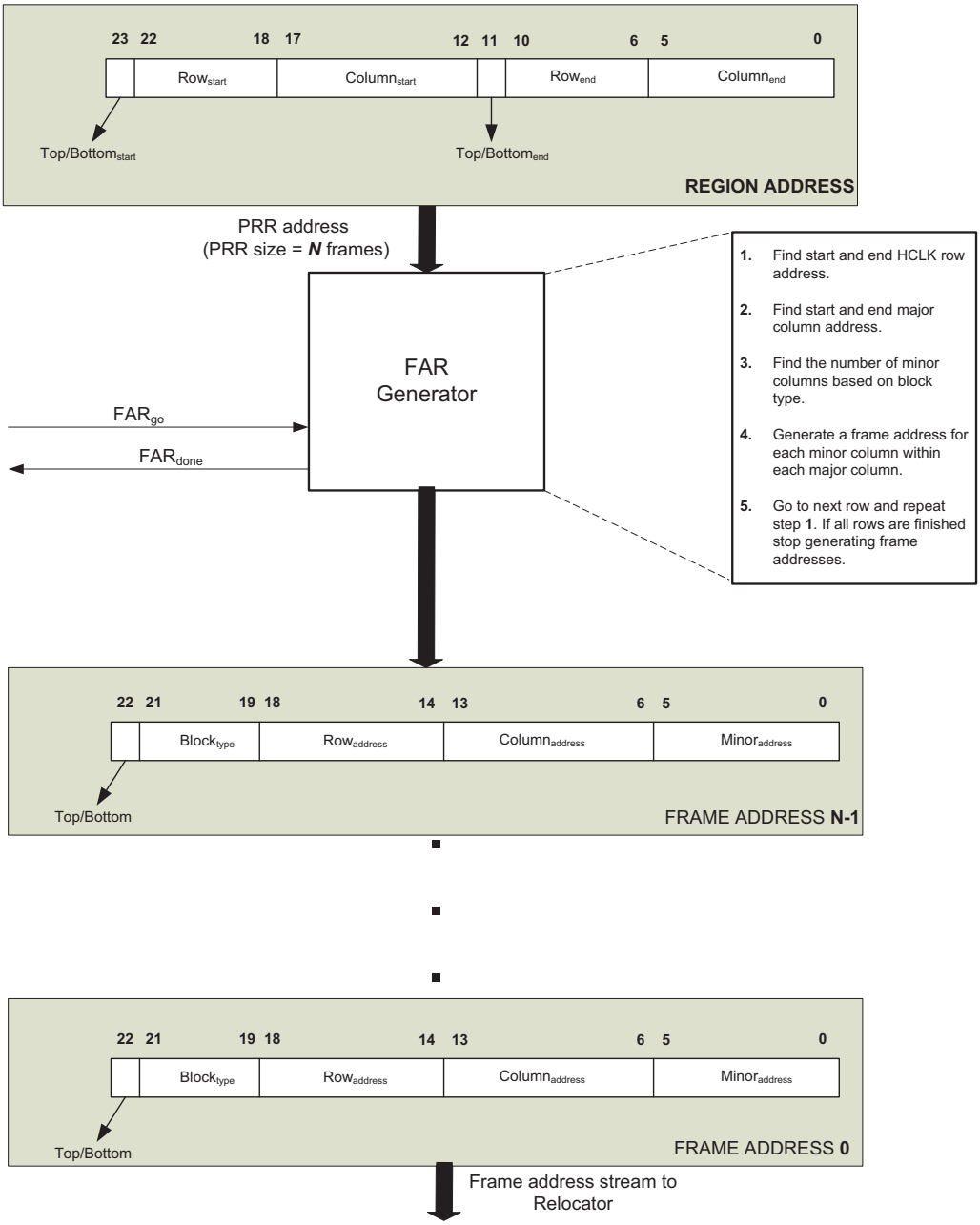

Fig. 5.4: FAR Generator converting region address to frame address stream.

it auto-increments the $FDB_{address}$ every time it receives the FDB_{go} signal is asserted high by the AMRC controller. For the current AMRC implementation, the FDB uses a BRAM storage with 16384 addressable locations that can store up to 399 frames, each 32-bit wide. However, the FDB can easily be interfaced with different BRAM sizes as well.

5.1.4 Relocator

The Relocator responds to three modes similar to the AMRC modes, (i) relocate, (ii) copy, and (iii) configure. The functionality of each Relocator mode is defined as follows.

1. Relocate Mode: The Relocator module uses the frame address pair generated by the FAR Generator (FAR_{Src} and FAR_{Dest}) to send a Read Command Sequence (RCS) to the ICAP wrapper for reading one frame from source PRR (using FAR_{Src}), followed by a Write Command Sequence (WCS) to the ICAP wrapper for writing that frame into the destination PRR (using FAR_{Dest}). After this it sends an acknowledgement signal ($Relocator_{done}$) to the AMRC controller, which triggers the generation of the next frame address pair from the FAR Generator. Figure 5.5 shows the state transition diagram for the Relocator in relocate mode. Figures 5.6 and 5.7 show the RCS and WCS sent to the ICAP wrapper, respectively.

2. Copy Mode: In copy mode, the Relocator's inputs are the source PRR's current frame address (FAR_{Src}) generated by the FAR Generator module and the BRAM address ($FDB_{address}$) generated by the FDB module. These are used to send a RCS to the ICAP wrapper for reading one frame from source PRR (using FAR_{Src}), followed by a write operation to the FDB. Once the frame has been copied to the FDB, the Relocator sends an acknowledgement signal ($Relocator_{done}$) to the AMRC controller which triggers the generation of the next frame address from the FAR Generator as well as the next BRAM address for storage using the FDB_{go} signal. Figure 5.8 shows the state transition diagram for the Relocator in copy mode.

3. Configure Mode: In configure mode, the Relocator's inputs are the BRAM address ($FDB_{address}$) generated by the FDB module and the destination PRR's frame address ($FAR_{Configure}$) from the AMRC controller module. These are used to read frames in bulk from the FDB, followed by a Bulk Write Command Sequence ($BWCS$) sent to the ICAP wrapper to configure the destination PRR (using $FAR_{Configure}$) with the frame data stored in the FDB. The Relocator performs the reconfiguration using bulk writes

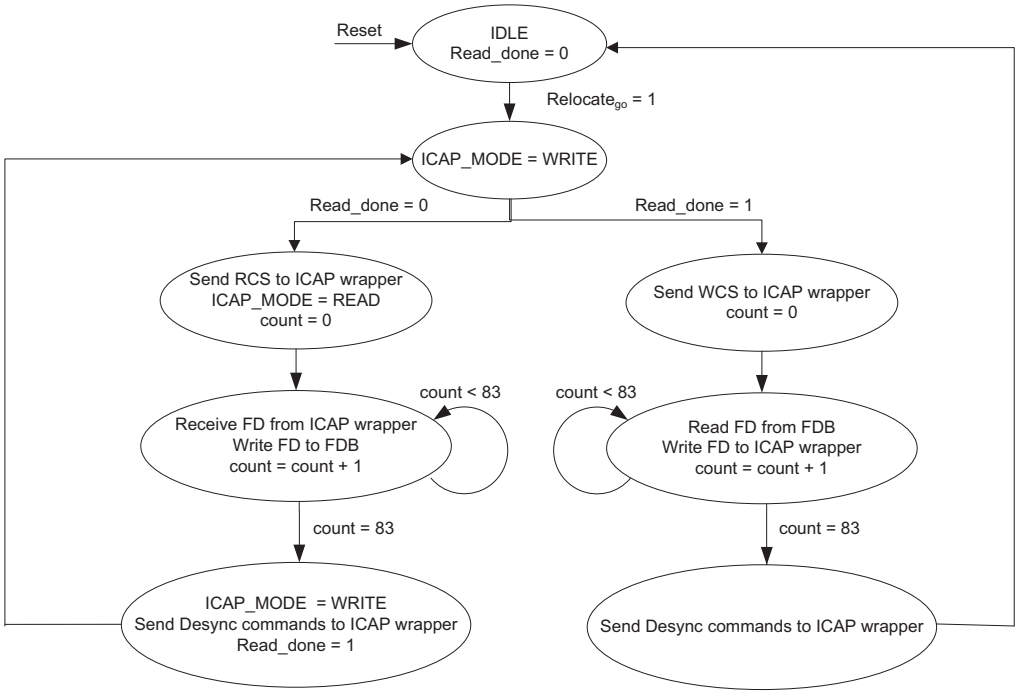

Fig. 5.5: Relocator state diagram for relocate mode.

of up to 44 frames depending on which resource type (CLB/DSP/BRAM) the frame data belongs to. Once the frame configuration is completed, the Relocator sends an acknowledgement signal ($Relocator_{done}$) to the AMRC controller, which triggers the generation of the next destination PRR's frame address as well as the next BRAM address to be read using the FDB_{go} signal. This process is repeated until all the frames in the FDB's BRAM storage have been written into the destination PRR. Figure 5.9 shows the state transition diagram for the Relocator in configure mode. Figure 5.10 shows the $BWCS$ sent to the ICAP wrapper in configure mode.

5.1.5 ICAP Wrapper

The ICAP wrapper is instantiated using Xilinx's internal ICAP device primitive (refer to discussion on ICAP in Sec. 2.4). The ICAP wrapper is driven by the input control signals and input data fed from the Relocator module. The output of this wrapper is sent to the memory (FDB) for storing the frame data coming from the ICAP.

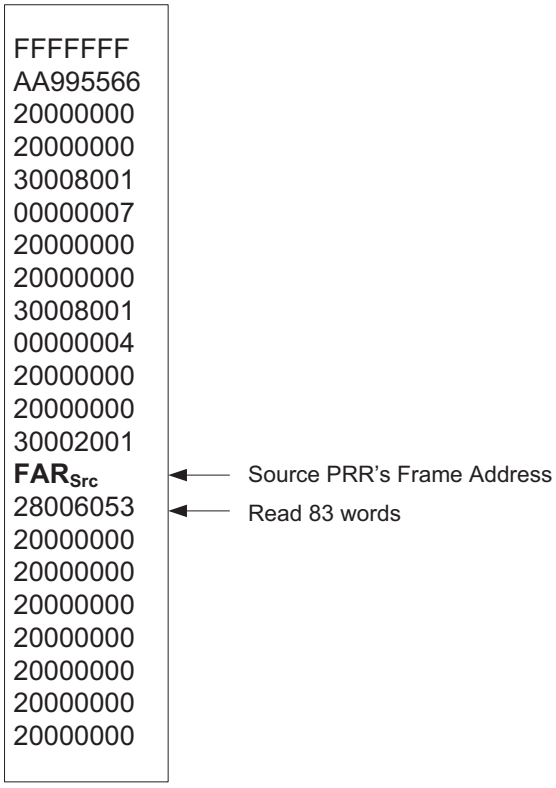

Fig. 5.6: Read Command Sequence (RCS).

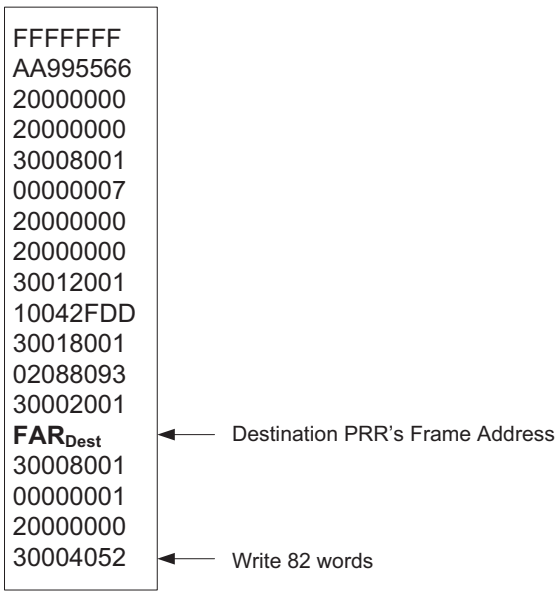

Fig. 5.7: Write Command Sequence (WCS).

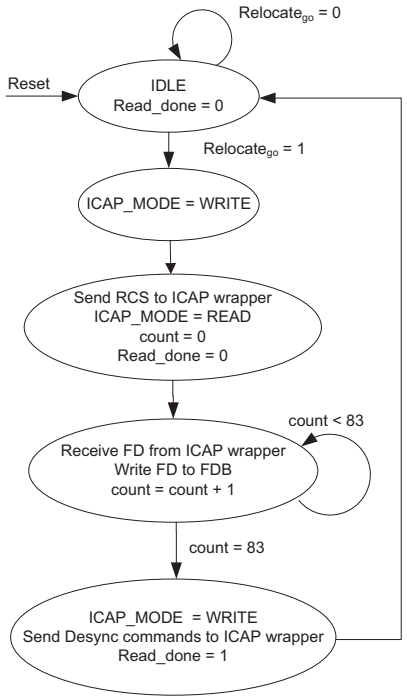

Fig. 5.8: Relocator state diagram for copy mode.

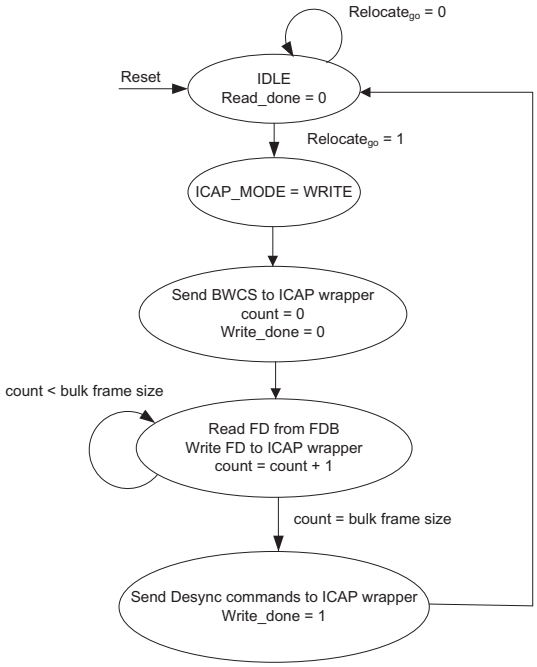

Fig. 5.9: Relocator state diagram for configure mode.

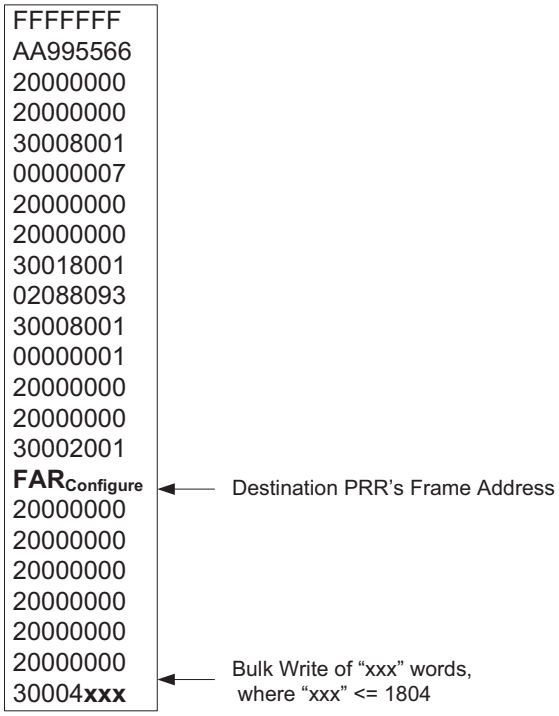

Fig. 5.10: Bulk Write Command Sequence (BWCS).

5.2 AMRC Performance Analysis

In this section, we compare the performance of ARC [6] and the proposed hardware implementation, AMRC. The performance analysis presented here is based on the performance estimation proposed in Sec. 4.2. The parameters are described in Table 4.1. Table 5.1 shows a comparison of the various latencies incurred during relocation between PRRs using PRR-PRR relocation, as compared to using M-FDR.

The ARC and AMRC are similar hardware techniques for PBR, which read and write the frame data from the active bitstream in PRRs. Therefore, the frame read and overheads given in parameters *1* through *5* in Table 5.1 are the same for both methods. Parameters *6* through *11* are the different latencies in reading a single frame from a PRR using both techniques. Parameter *12* ($\boldsymbol{T}_{readFD}$), is the sum total of all the latencies given in parameters *6* through *11*. We observe that the value of T_{readFD} is equal for both the techniques. This is because both the AMRC and ARC read PRRs frame by frame, and therefore both methods need to account for the per frame overhead involved.

Table 5.1: Performance analysis of AMRC versus ARC.

	Parameter #	Parameter Name	Number of words	AMRC (M-FDR) (number of cycles)	ARC (PRR-PRR relocation [6]) (number of cycles)
	1	T_{gen}^{α}	x	1	1
	2	$T_{overheadW}$	n/a	4	4
	3	$T_{overheadR}$	n/a	4	4
	4	$T_{write}(x)$	x	x	x
	5	$T_{read}(x)$	x	x	x
Single Frame Read	6	$T_{gen}^{syncRdCmds}$	18	1	1
	7	$T_{writeICAP}^{syncRdCmds}$	18	22	22
	8	$T_{readICAP}^{FD}$	83	87	87
	10	$T_{gen}^{desyncCmds}$	4	1	1
	11	$T_{writeICAP}^{desyncCmds}$	4	8	8
	12	$\boldsymbol{T_{readFD}}$	**n/a**	**119**	**119**
Bulk Frame Writes (440 frames)	13	$T_{gen}^{syncWrCmds}$	24	1	1
	14	$T_{writeICAP}^{syncWrCmds}$	18	22	22
	15	$T_{writeICAP}^{FD}$	18040	18044	n/a
			36080	n/a	36084
	16	$T_{gen}^{desyncCmds}$	4	1	1
	17	$T_{writeICAP}^{desyncCmds}$	4	8	8
	18	$Factor_{WriteOverhead}$	n/a	146	440
	19	$\boldsymbol{T_{writeFD}}$	**n/a**	**22716**	**50164**
	20	$T_{bitReversal}$	18040	0	0
	21	$T_{overall}$	n/a	75076	102524

Parameters *13* through *17* are the latencies involved in writing a payload of 440 frames (20 CLB columns) to a PRR using both the AMRC and ARC. Parameter *18* ($Factor_{WriteOverhead}$) is the factor of write overhead involved when writing 440 frames using both techniques. For AMRC we can observe that this is relatively low, when compared to ARC where it is 440 which is simply the total number of frames being written. This shows the performance advantage in writing frames in bulk using AMRC, since the frame overhead is for every two CLB/DSP/BRAM columns written to the PRR. In the case of relocation to the opposite half of the FPGA both methods have no overhead because the frame data is bit-flipped on-the-fly without incurring any processing delays by both the AMRC and ARC. The overall time ($T_{overall}$) is the time to read and write 440 frames using both these techniques. We find that the AMRC is 26.77% faster than ARC in terms of overall relocation time and 54.7% faster when performing bulk frame data writes to a PRR.

Chapter 6

Results

The hardware implementation for the M-FDR technique (AMRC) was implemented and tested on a Xilinx Virtex 4 SX35 FPGA based ML402 evaluation platform [34] clocked at 100 MHz. The Xilinx ISE tool flow was used to synthesize, map, place, and route both the AMRC design, as well as the PRMs. All PRR-PRR relocations and M-FDR test scenarios were implemented using the Xilinx EAPR tool flow [11]. AMRC performance is compared against ARC [6], BiRF [4], and the software-based relocation approach proposed by Carver et al. [5] based on the information provided in their respective publications. The test cases can be categorized into two subsets as follows:

1. PBR Tests: The objective here is to show a comparison in terms of the relocation time achieved using different methods for relocating single and multiple row PRRs. The results are discussed in Sec. 6.1.

2. Run-time Parallelization Tests: The objective here is to demonstrate support for dynamically increasing and decreasing the parallelism of various PRMs, for a given number of PRRs available. The results are discussed in Sec. 6.2.

6.1 PBR Test Cases and Results

In this section we compare the performance of ARC and AMRC. The performance results for the different test cases are given in Table 6.1. Test cases 2, 4, 7, 9, and 12 use multi-row PRRs that are two rows tall. These test cases are further sub-divided based on (i) type of design (systolic array based processing element (PE) or non-systolic array based), and (ii) DSP48 [35] usage.

It can be observed from Table 6.1 that the AMRC relocation is faster than the ARC relocation for all the test cases. There is an average performance increase of 26.6% when

Table 6.1: Relocation time (in ms) comparison between the AMRC and other approaches. DWT stands for Discrete Wavelet Transform.

Id	Test case	# frames	Frame data size (bytes)	Bitstream size (bytes)	# BRAMs (required by BiRF [4] and Carver et al. [5])	AMRC [1]	ARC [2] [6]	BiRF [4]	Carver et al. [5]	
						Same/Opp half	Same/Opp half	Same half	Same half	Opp half
Systolic Array PE (not using DSP48)										
1	Faddeev [7]	195	31980	31159	14	0.36	0.48	84.7	3.38	8.86
2	Faddeev [7] (Multirow)	258	42312	35557	18	0.47	0.64	96.7	3.86	10.11
3	DWT [7]	195	31980	30693	14	0.36	0.49	83.4	3.33	8.73
4	DWT [7] (Multirow)	258	42312	36717	18	0.47	0.64	99.8	3.98	10.44
5	Matrix Multiplier	432	70848	68469	30	0.79	1.08	186.1	7.42	19.47
Systolic Array PE (using DSP48)										
6	Faddeev [7]	195	31980	32349	15	0.36	0.49	87.9	3.5	9.2
7	Faddeev [7] (Multirow)	258	42312	40537	20	0.47	0.64	110.2	4.4	11.53
8	DWT [7]	195	31980	33045	15	0.36	0.49	89.8	3.58	9.39
9	DWT [7] (Multirow)	258	42312	41505	21	0.47	0.64	112.8	4.5	11.8
10	Matrix Multiplier	432	70848	65261	29	0.79	1.08	177.3	7.07	18.56
Non-Systolic Array implementation (using DSP48)										
11	DWT [7]	303	49692	47897	21	0.55	0.76	130.2	5.19	13.62
12	DWT [7] (Multirow)	346	56744	52553	26	0.63	0.86	142.9	5.7	14.94

[1]BRAM usage for AMRC is fixed and mentioned in Table 6.2.
[2]BRAM usage for ARC is fixed and mentioned in Table 6.2.

using AMRC as compared to ARC. The average speedup of AMRC compared to BiRF is approximately 231, while compared to the method proposed by Carver et al. [5] the average speedup is 9.2 for the circuits on the same side and 24.2 for circuits on the opposite side. The primary reason for a large speedup for the AMRC when compared to both methods is because BiRF needs to get the partial bitstream for modification from off-chip memory, and Carver et al. [5] perform relocation in software. Both AMRC and ARC operate on the frame data instead of changing the partial bitstream, however the AMRC relocates faster because of bulk frame writes to PRRs. The frame data size is larger when compared to the bitstream size in Table 6.1 because the PlanAhead [16] placement and routing tool typically compresses multiple identical frames into a single frame, thus reducing the partial bitstream's overall size. There is insufficient information about relocation to the opposite half using BiRF.

The resource estimates of ARC and the comparisons are shown in Table 6.2. The footprint shown is on a Virtex-4 SX35 chip. To estimate the number of BRAMs, the

Microblaze is instantiated with 64KB of memory. From Table 6.2 it can be observed that the AMRC has a higher on-chip memory usage compared to other methods, in order to copy and configure real-world designs onto the PRRs. However, with later Xilinx FPGA families like the Virtex-5 and Virtex-6 the total number of BRAMs is several times larger and each BRAM block at least twice the size of a typical BRAM block on a Virtex-4 FPGA. This means the cost in terms of BRAMs for the AMRC will become negligible on larger FPGA chips.

6.2 Run-time Parallelization Test Cases and Results

In this section we demonstrate the applicability of AMRC to run-time parallelization of PRMs. The following setup is used for this exercise:

1. Four designs are taken consisting of two adders, one subtracter, and one multiplier.

2. Four identical PRRs are created each two CLB columns wide, and two HCLK rows tall. Table 6.3 summarizes the layout of each of the PRRs.

3. The initial configuration of the PRRs has the notation, *AASM empty*, where *A* refers to adder, *S* refers to subtracter, and *M* refers to multiplier. The term *empty* refers to the state of the FDB in the AMRC. The term empty implies that the FDB does not contain any frame data.

Figure 6.1 shows the four PRRs, and their initial placement on a Xilinx Virtex 4 SX35 FPGA floorplan. Consider an application where at any given point in time, only two of the three designs are in use, and either the adder and multiplier or the subtracter and multiplier are active. Figure 6.2 shows the test case state machine with three states, (i) initial setup, (ii) state A where the adder is parallelized, and (iii) state S where the subtracter is parallelized.

During dynamic parallelization the system is setup in the initial state, which can have any PRR configuration as long as there is one copy of each PRM to start with. The startup overhead is the time to switch from the initial state to *state A* or *S*. This is a one-time

Table 6.2: Resource requirements of AMRC, ARC, and BiRF (Microblaze is instantiated with 64KB of memory).

Relocation Architecture	**LUTs**	**FFs**	**BRAMs**
AMRC (with Microblaze)	3041	4139	91
AMRC (with state machine)	1393	3099	59
ARC [6] (with Microblaze)	2787	1928	33
ARC [6] (with state machine)	1072	686	1
BiRF [4]	2047	1574	32

Table 6.3: Layout of PRRs for testing on the FPGA.

PRR	**# Major Column Start**	**Major Column End**	**HCLK Row Start**	**HCLK Row End**
0	6	7	2	1
1	15	16	1	0
2	30	31	2	1
3	39	40	1	0

overhead because once we are either in *state A* or *S*, we only context switch between these two states based on the user's request for parallelization. For example, suppose the system is in *state A*, i.e., two PRRs contain adder circuits and two contain multiplier circuits. If the user now makes a request to re-parallelize the subtracter, the subtracter's frame data in the FDB (shown within {} in Fig. 6.2) is configured into the PRRs. We then copy the frame data belonging to the adder into the FDB again. Finally, we relocate the PRR containing the multiplier, to generate two PRRs with multipliers again. The average time to context switch between the adder (*state A*) and subtracter (*state S*) was found to be 0.37 ms. This provides a good estimate of how much switching overhead to expect when designing a run-time parallelized design. The switching time increases with the number of frames occupied by a PRR. Due to resource limitations on the Virtex 4 SX35 FPGA, we were unable to test circuits with more than four frames. However, run-time parallelization using the AMRC can be easily extended to larger FPGAs.

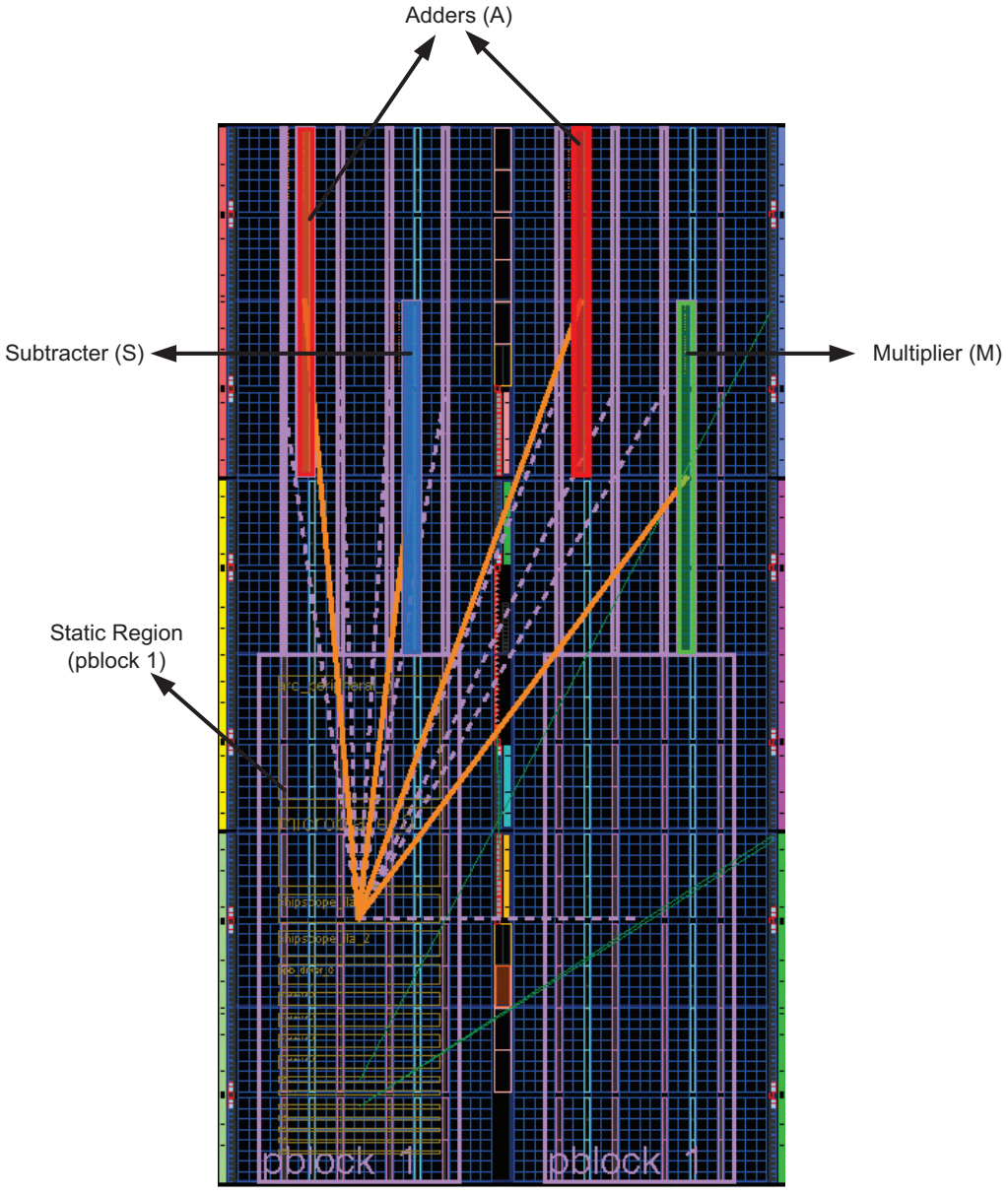

Fig. 6.1: Initial PRR placement in {A,S,A,M} arrangement for AMRC run-time parallelization tests.

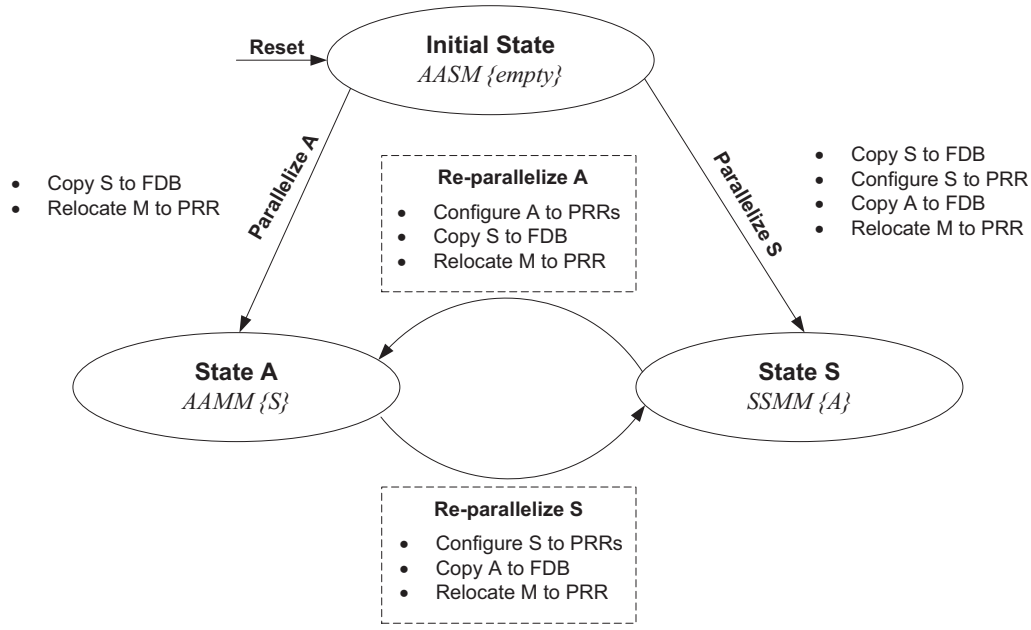

Fig. 6.2: Run-time parallelization state machine for testing.

Chapter 7

Conclusion and Future Work

In this thesis, existing approaches to bitstream relocation on FPGAs and their drawbacks have been discussed. This work proposes a novel multi-row M-FDR algorithm that can relocate frame data between PRRs faster than contemporary state-of-the-art relocation techniques by Corbetta et al. [4], Carver et al. [5], and Sudarsanam et al. [6]. The proposed algorithm has been implemented using a dedicated hardware architecture called the Accelerated Memory-based Reconfiguration Circuit (AMRC). A performance estimation model of the AMRC has been proposed that quantifies the performance benefit of AMRC over ARC [6], which was found to be the fastest relocation method among previous approaches that were studied. AMRC was found to be on an average 26.6% faster than ARC [6] when relocating PRRs. An average speedup of 231x over BiRF [4], while for the method proposed by Carver et al. [5] a speedup of 9.2x for circuits on the same side of the FPGA, and 24.2x for circuits on the opposite side of the FPGA was observed.

The major challenge in this thesis involved understanding the existing relocation techniques like PRR-PRR relocation [6] and BiRF [4] and identifying their drawbacks. Several tests were used to help identify the performance bottlenecks when using the ICAP and off-chip versus on-chip memory during frame data relocation. Another important constraint to manage during M-FDR is that the ICAP cannot be written more than 2048 frame data words at once. Therefore for larger designs, a column pattern had to be identified for copying frames in groups of either two CLB columns, or one CLB and one DSP column, or one BRAM column, because of the way columns are addressed and arranged in Xilinx Virtex-4 FPGAs. Bit reversal is an important consideration when moving frames to the opposite side of the FPGA. The AMRC has been designed to handle configuration requests to any side of the FPGA and to reverse bits for only those frames that fall on the opposite side of

the FPGA.

Future research can be carried out in the following directions.

1. Use compression techniques to reduce the memory footprint of frame data in the FDB for efficient on-chip memory utilization.

2. Implement the AMRC on FPGAs with higher gate densities like the Virtex-5 and Virtex-6 families of Xilinx FPGAs to reduce the penalty of using on-chip memory to store frame data.

3. Improve the AMRC hardware to store more than one circuit in the FDB, so as to support a more granular form of run-time parallelization between PRMs.

4. Implement sharing of the FDB between the static and reconfigurable regions of the SoC so that unused BRAM memory in the FDB can be re-used by the static region for storage.

5. Investigate ways to further improve the relocation time, such as reducing relocation overhead during the frame read process, and implementing bulk frame reads.

References

[1] T. Becker, W. Luk, and P. Y. K. Cheung, "Enhancing relocatability of partial bitstreams for runtime reconfiguration," in *Proceedings IEEE International Symposium on Field-Programmable Custom Computing Machines*, pp. 35–44, 2007.

[2] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, p. 151. Washington, DC: IEEE Computer Society, 2005.

[3] H. Kalte and M. Porrmann, "Replica2pro: task relocation by bitstream manipulation in virtex-ii/pro fpgas," in *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pp. 403–412. New York: Association for Computing Machinery, 2006.

[4] S. Corbetta, M. Morandi, M. Novati, M. Santambrogio, D. Sciuto, and P. Spoletini, "Internal and external bitstream relocation for partial dynamic reconfiguration," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, vol. 17, no. 11, pp. 1650–1654, Nov. 2009.

[5] J. Carver, R. Pittman, and A. Forin, "Relocation and automatic floor-planning of fpga partial reconfiguration bitstreams," *Microsoft Research Technical Report*, Aug. 2008.

[6] A. Sudarsanam, R. Kallam, and A. Dasu, "Prr-prr dynamic relocation," *IEEE Computer Architecture Letters*, vol. 8, pp. 44–47, July 2009 [Online]. Available: `http://dx.doi.org/10.1109/L-CA.2009.49`.

[7] A. Sudarsanam, R. Barnes, J. Carver, R. Kallam, and A. Dasu, "Dynamically reconfigurable systolic array accelerators: A case study with extended kalman filter and discrete wavelet transform algorithms," *Computers Digital Techniques, The Institution of Engineering and Technology (IET))*, vol. 4, no. 2, pp. 126–142, Mar. 2010.

[8] A. Sreeramareddy, J. G. Josiah, A. Akoglu, and A. Stoica, "Scars: Scalable self-configurable architecture for reusable space systems," in *AHS '08: Proceedings of the 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 204–210. Washington, DC: IEEE Computer Society, 2008.

[9] 1-CORE Technologies, "Fpga logic cells comparison," `[http://www.1-core.com/library/digital/fpga-logic-cells/]`, 2004 - 2009.

[10] Xilinx, "Design flow overview," `[http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev0013_5.html]`, 2005.

[11] Xilinx, "Early access partial reconfiguration user guide," UG208 (v1.1), Xilinx Inc., Mar. 6, 2006.

[12] C. Kao, "Benefits of partial reconfiguration," *Xcell Journal*, pp. 65–76, 2005.

[13] Xilinx, "Edk concepts, tools, and techniques a hands-on guide to effective embedded system design," [http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/edk_ctt.pdf], 2009.

[14] Xilinx, "Microblaze soft processor core," [http://www.xilinx.com/tools/microblaze.htm], 2010.

[15] C. Conger, R. Hymel, M. Rewak, A. George, and H. Lam, "Fpga design framework for dynamic partial reconfiguration," in *Proceedings of the 15th Reconfigurable Architecture Workshop, RAW 2008*, pp. 1–8, April 14-15, 2008.

[16] Xilinx, "Planahead user guide," [http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/PlanAhead_UserGuide.pdf], 2009.

[17] Xilinx, "Virtex-4 configuration guide," [http://www.xilinx.com/support/documentation/user_guides/ug071.pdf], 2009.

[18] D. Mesquita, O. Moraes, J. Palma, R. Mller, and N. Calazans, "Remote and partial reconfiguration of fpgas: tools and trends," in *Proceedings of International Parallel and Distributed Processing Symposium*, pp. 22–26, 2003.

[19] E. Carvalho, N. Calazans, E. Briao, and F. Moraes, "Padreh - a framework for the design and implementation of dynamically and partially reconfigurable systems," in *Proceedings of the 17th Symposium on Integrated Circuits and Systems Design, Simposio Brasileiro de Concepcao de Circuitos Integrados (SBCCI)*, pp. 10–15, 2004.

[20] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Norwell, MA: Kluwer Academic Publishers, 2002.

[21] F. Ferrandi, M. Santambrogio, and D. Sciuto, "A design methodology for dynamic reconfiguration: the caronte architecture," in *Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium*, p. 4, 2005.

[22] A. Raghavan and P. Sutton, "Jpg - a partial bitstream generation tool to support partial reconfiguration in virtex fpgas," in *Proceedings of International Parallel and Distributed Processing Symposium, IPDPS 2002, Abstracts and CD-ROM*, pp. 155–160, 2002.

[23] S. Guccione, D. Levi, and P. Sundararajan, "Jbits: Java based interface for reconfigurable computing." Xilinx Inc., 1999.

[24] K. Nasi, T. Karouhalis, M. Danek, and Z. Pohl, "Figaro - an automatic tool flow for designs with dynamic reconfiguration," in *International Conference on Field Programmable Logic and Applications*, pp. 590–593, 2005.

[25] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in virtex fpgas," in *IEE Proceedings of Computers and Digital Techniques*, vol. 153, no. 3, pp. 157–164, Feb. 2006.

[26] O. Diessel and G. Milne, "Hardware compiler realising concurrent processes in reconfigurable logic," in *IEE Proceedings of Computers and Digital Techniques*, vol. 148, no. 45, pp. 152–162, July/Sept. 2001.

[27] A. Bailey, "Using the circal process algebra in digital system design," in *IEE Colloquium on Formal and Semi-Formal Methods for Digital Systems Design*, pp. 4/1–4/4, 1991.

[28] M. Gericota, G. Alves, M. Silva, and J. Ferreira, "Run-time management of logic resources on reconfigurable systems," in *Design, Automation and Test in Europe Conference and Exhibition*, pp. 974–979, 2003.

[29] H. Tan and R. F. DeMara, "A physical resource management approach to minimizing fpga partial reconfiguration overhead," in *IEEE International Conference on Reconfigurable Computing and FPGAs*, pp. 1–5, Sept. 2006.

[30] L. Singhal and E. Bozorgzadeh, "Physically-aware exploitation of component reuse in a partially reconfigurable architecture," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2006*, p. 8, 2006.

[31] E. Horta, J. Lockwood, D. Taylor, and D. Parlour, "Dynamic hardware plugins in an fpga with partial run-time reconfiguration," in *Proceedings of the 39th Design Automation Conference*, pp. 343–348, 2002.

[32] Y. Krasteva, E. de la Torre, T. Riesgo, and D. Joly, "Virtex ii fpga bitstream manipulation: Application to reconfiguration control systems," in *International Conference on Field Programmable Logic and Applications*, pp. 1–4, 2006.

[33] D. Montminy, R. Baldwin, P. Williams, and B. Mullins, "Using relocatable bitstreams for fault tolerance," in *2nd NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 701–708, 2007.

[34] Xilinx, "Ml40x getting started tutorial," `[http://www.xilinx.com/support/documentation/boards_and_kits/ug083.pdf]`, 2006.

[35] Xilinx, "Xtremedsp for virtex-4 fpgas," `[http://www.xilinx.com/support/documentation/user_guides/ug073.pdf]`, 2008.